TURING 图灵程序设计丛书



Cassandra

权威指南

Cassandra: The Definitive Guide

[美] *Eben Hewitt* 著

Jonathan Ellis 序

王旭 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



图灵程序设计丛书

Cassandra权威指南

Cassandra: The Definitive Guide

[美] Eben Hewitt 著
Jonathan Ellis 序
王 旭 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

Cassandra权威指南 / (美) 休伊特 (Hewitt,E.) 著;
王旭译. -- 北京 : 人民邮电出版社, 2011.8
(图灵程序设计丛书)
书名原文: Cassandra : The Definitive Guide
ISBN 978-7-115-25854-0

Ⅰ. ①C… Ⅱ. ①休… ②王… Ⅲ. ①关系数据库—数据库管理系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2011)第129426号

内 容 提 要

本书是一本广受好评的 Cassandra 图书。与传统的关系型数据库不同，Cassandra 是一种开源的分布式存储系统。书中介绍了它无中心架构、高可用、无缝扩展等引人注目的特点，讲述了如何安装、配置 Cassandra 及如何在其上运行实例，还介绍了对它的监控、维护和性能调优手段，同时还涉及了 Cassandra 相关的集成工具 Hadoop 及其类似的其他 NoSQL 数据库。

本书适合数据库开发人员与网站开发者阅读。

图灵程序设计丛书
Cassandra 权威指南

◆ 著 [美] Eben Hewitt
序 [美] Jonathan Ellis
译 王 旭
责任编辑 傅志红
执行编辑 李 盼

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：19
字数：394千字 2011年8月第1版
印数：1－3 500册 2011年8月北京第1次印刷
著作权合同登记号 图字：01-2011-0812号
ISBN 978-7-115-25854-0

定价：59.00元
读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”，创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会聚集了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

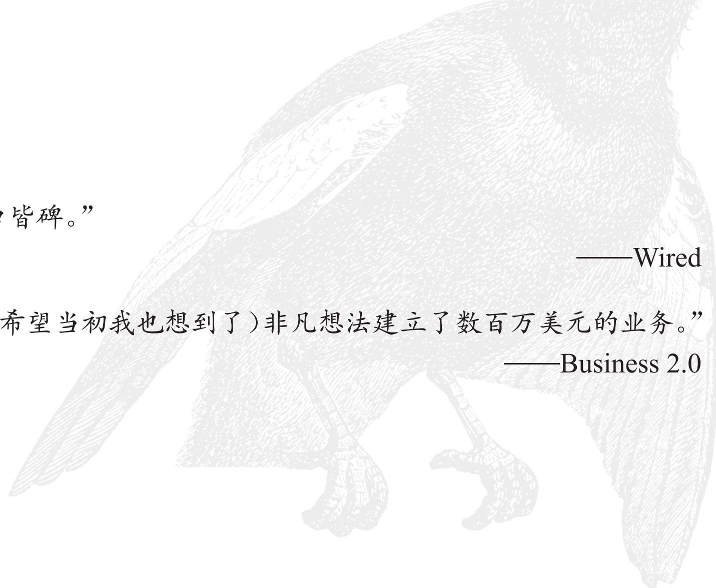
业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0



“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

译者序

对于一位分布式存储系统的开发者，Cassandra 无疑是非常引人注目的，它的无中心架构、高可用性、无缝扩展等继承自亚马逊 Dynamo 的特质，相对于其他主从架构的 NoSQL 系统更加简洁，也更具有美感。

我从 2010 年初开始关注这个系统，并翻译过几篇 Cassandra 相关的文章，还引起一些读者热烈的讨论。2010 年底，当刘江老师为本书寻找译者时，我按捺不住，毛遂自荐，并随后在 2011 年 1 月中下旬开始了本书的翻译工作。我用了三个月的业余时间，终于在 4 月份完成了译稿。因为 Cassandra 仍在快速开发中，翻译时我也尽力争取快一些，以便能让中文版出版时不至于落伍。

本书对 Cassandra 的概念、架构、配置、使用进行了全面的介绍，非常详尽，而且给出了很多参考信息。对于希望了解 Cassandra、评估 Cassandra 是否是适合自己的应用，以及开始着手在 Cassandra 上进行应用开发的人都是不错的读物。当然，如果想参与 Cassandra 的开发或做更深入的工作，还需要直接通过源代码来获取更详尽的信息。

在翻译中，我尽力使用已有的、被广泛接受的名词或术语，对于一些译法没有被广泛接受的术语，在不产生歧义的前提下，我会选择一个自以为恰当的词，有时还会给出英文，以避免读者不能将代码和本书给出的名词对应上。还有一些名词尚没有贴切的中文译法，或是译出容易产生歧义，或是国内开发者已习惯使用英文，这时我在翻译中保留了英文原文。这些选择都以帮助理解、避免歧义为首要考虑。

本书的翻译工作得到了很多朋友和网友的关注，希望没有让他们久等。我的同事郭磊涛，作为数据库和 HBase 的专家、Cassandra 用户，在本书的翻译过程中给予了很多

有益的帮助。感谢现在 CSDN 的刘江老师，给我这个机会把 Cassandra 介绍给大家。当然，还要感谢图灵的编辑杨海玲、傅志红，还有李松峰在本书的翻译过程中做了大量的细心工作。

希望本书的翻译出版能对读者进入 NoSQL 的世界、开始自己的 Cassandra 应用有些许的帮助。

序

Cassandra 是 Facebook 于 2008 年 7 月开源的项目。它最早的版本主要是由一位亚马逊前雇员和一位微软的工程师写成的。这个系统受到了亚马逊前卫的键 / 值存储系统 Dynamo 的巨大影响。Cassandra 实现了 Dynamo 风格的副本复制模型和没有单点失效的架构，但增加了更为强大的“列族”数据模型。

当年 12 月，在 Rackspace 要求我帮他们建立一个可扩展的数据库的时候，我加入到这个项目之中。那是个很好的时机，因为今天所有重要的开源可扩展数据库在那时都有了，可以做做比较。尽管最初 Cassandra 只有一个主要的应用案例，但它的底层架构是最强大的，于是，我致力于改进代码，同时建立一个社区。

之后，Cassandra 被接纳为 Apache 的孵化器项目，并于 2010 年 3 月毕业成为顶级项目。此时它已经成为了一个真实的开源软件的成功案例，Rackspace、Digg、Twitter 等公司都成了忠实的用户，他们不愿意从零开始写自己的数据库，但却希望一起来构建一个更优秀的系统。

今天的 Cassandra 已经远不止是当初那个（现在也还在）用来驱动 Facebook 的收件箱搜索的系统了，按照 Tony Bain 的说法，它已经成为了“事务处理性能的不二赢家”，而且在可靠性和可扩展性方面具有显赫的声誉。

随着 Cassandra 逐渐成熟并获得了更多的主流用户，我们显然有为它提供商业支持的需要，于是，Matt Pfeil 和我在 2010 年 4 月共同创立了 Riptano。帮助推动 Cassandra 的应用具有丰富的回报，特别是可以看到更多的还没有被公开讨论过的应用。

另一个需求就是一本关于 Cassandra 的书。和很多开源项目一样，Cassandra 的文档一直就是一个弱项。而且即使是文档最终得到了改善，一本这样的书仍然会非常有用。

感谢 Eben 来承担这项集艺术与科学于一身的艰巨任务，讲解 Cassandra 的开发与部署。读者朋友现在有机会可以有条理地学习这些新概念了。

——Jonathan Ellis

Apache Cassandra 项目主席、Riptano 联合创始人

前言

选择Apache Cassandra

Apache Cassandra 是一个自由、开源的分布式数据存储系统，与传统的关系型数据库管理系统截然不同。

Cassandra 在 2009 年 1 月成为了 Apache 基金会的一个孵化器项目。不久，以 Apache Cassandra 项目主席 Jonathan Ellis 为首的开发者们发布了 Cassandra 0.3，随后稳定不断地发布新的小版本。虽然 Cassandra 在本书完成时仍然没有达到 1.0 发布版本，但已经被互联网领域的很多巨头使用在了生产系统之中，他们包括 Facebook、Twitter、Cisco、Rackspace、Digg、Cloudkick、Reddit 等。

因为它非常出色的技术特性，Cassandra 已经变得非常受欢迎了。它具有持久性、无缝扩展性、可调的一致性。它的写操作非常快，可以存储上百 TB 数据，而且是无中心的和对称的，所以不会有单点失效。它还是高度可用的，提供了无 schema 的数据模型。

目标读者

本书适用于各类读者。它对以下读者都会非常有用。

- 大规模、高容量网站的开发者，比如 Web 2.0 的社交应用。
- 需要理解这个高性能、无中心、弹性数据存储系统的应用架构师或数据架构师。
- 希望理解如何实现容错、最终一致的数据存储系统的标准关系型数据库系统管理员或开发者。

- 希望了解 Cassandra 的优势（和不足）以及其他相关的列数据库，以帮助进行技术路线选择的管理者。
- 正在进行 Cassandra 或其他非关系型数据库相关项目的学生、分析师或研究员。

本书是一本技术指南。从某种意义上说，Cassandra 代表了一种对数据的新的思考。在过去的 15 ～ 20 年间，很多合格的职业开发者都在使用纯粹的关系型或是面向对象的术语来描述他们的数据。Cassandra 的数据模型与此非常不同，起先可能很难吸引你，特别是对于数据库（应该）是什么已经有了先入为主的概念的人，更是如此。

使用 Cassandra 并不意味着你必须成为一个 Java 开发者。不过，Cassandra 是用 Java 开发的，所以若要深入分析源代码，你需要对 Java 语言有更坚实的理解。虽然不一定需要懂得 Java，但 Java 可以帮助你更好地了解异常、学会如何编译源码以及使用一些流行的客户端。本书中的很多例子都是用 Java 写成的。尽管如此，因为 Cassandra 使用了语言中立的 RPC 接口，所以你可以使用多种语言来开发 Cassandra 应用，包括 C#、Scala、Python 以及 Ruby 等。

最后，本书假设读者已经了解了 Web 是如何工作的，能够使用集成开发环境，并对数据驱动的应用的典型问题有某些了解。你可能是一个经验丰富的开发者或管理员，但是对于在 Cassandra 的世界里使用到的工具可能偶尔也不是非常熟悉。比如 Cassandra 使用 Apache Ivy 进行编译，而用一个流行的客户端（Hector）使用 Git 进行版本管理。当我感到你可能需要自己进行一些设置才能运行一个例子的时候，我会尽量予以说明。

本书的结构

本书把每章合理地设计为一个个独立的指南。因为本书是介绍 Cassandra 的，读者们可能背景各异，而且技术变化很快，所以这么处理非常重要。借用一个软件界的说法，我希望本书能够有点儿“模块化”。如果你是一个 Cassandra 新人，那么可以按照顺序阅读；而如果你已经有所了解，不需要介绍了，那么也可以在后面的章节里找到有价值的内容，把它们当做独立的指南来看。

本书的具体结构是这样的。

- 第 1 章 Cassandra 概况
  这一章介绍了 Cassandra，并讨论了它与众不同的特质、优势和目前的用户。

- 第 2 章 安装 Cassandra
  在这一章中，作者会带你在不同平台上安装 Cassandra。

- 第 3 章 Cassandra 的数据模型
  这里，我们介绍了 Cassandra 的数据模型以了解 Cassandra 中的列、超级列、行都是什么。我们特别介绍了 Cassandra 和传统的关系型数据库之间的差别。
- 第 4 章 应用实例
  这一章给出了一个完整可用的例子，将一个大家熟悉的领域中的应用实例从关系模型迁移到了 Cassandra 的数据模型之上。
- 第 5 章 Cassandra 的架构
  这一章会帮你理解在 Cassandra 进行读写操作时，到底都发生了什么，这个数据库是如何做到它的那些特点的，比如持久性和高可用性。我们深入到底层来了解一些更复杂的内部工作机制，比如 gossip 协议、提示移交、读时修复、Merkle 树等。
- 第 6 章 配置 Cassandra
  这一章介绍了如何设置分区器、副本放置策略和 snitch。我们配置了一个集群，了解不同配置选项对于集群的影响。
- 第 7 章 读写数据
  这是我们一直期待的时刻。这里介绍了 Cassandra 模型在查询和更新数据时与传统关系型数据库的不同，然后还使用 API 进行了操作。
- 第 8 章 客户端
  第三方开发者为 Cassandra 开发了很多不同的客户端，支持多种语言，包括 Java、C#、Ruby、Python 等，对 Cassandra 的底层 API 进行了再次抽象。我们会帮你从整体上了解这些客户端，这样你就可以选择一个适合自己的了。
- 第 9 章 监控
  一旦集群已经配置好并开始运行了，就需要监控它的利用率、内存占用和线程状况，了解它的日常行为。Cassandra 内建了丰富的 Java 管理扩展（JMX）接口，我们可以监控所有这些信息，甚至更多。
- 第 10 章 维护
  通过服务器自带的一些工具，可以更简单地进行很多 Cassandra 集群的日常维护工作。我们会看到如何退服一个节点，对集群进行负载均衡，获取统计信息以及进行其他日常维护操作任务。
- 第 11 章 性能调优
  Cassandra 的一个最值得一提的特性就是它的速度——非常地快。但有很多东西，包括内存设置、数据存储、硬件选择、缓存和缓冲区大小等，都需要进一步调优，从中获得更高的性能。

- 第 12 章 集成 Hadoop

  这一章由 Jeremy Hanna 写作。在这一章我们会把 Cassandra 放到一个更大的背景中，学习如何将它与 Hadoop 集成在一起，Hadoop 是 Google 的 Map/Reduce 算法目前一个十分流行的实现。

- 附录

  很多新的数据库都在今日海量数据的需求之下应运而生了，有的从“无 schema”模型中获益，有的支持更新的一些趋势，如语义网络。这里我们把 Cassandra 放到各种流行的非关系型数据库背景之中，分别了解面向文档的数据库、分布式哈希表、图数据库等，来更好地理解 Cassandra 所提供的东西。

- 词汇表

  理解一些确实很新的东西是相当困难的，Cassandra 中有些名词对于关系型应用的开发者和 DBA 来说可能非常陌生，我编写了一个词汇表，来方便大家阅读本书。如果某个概念让你不知所云，可以翻到词汇表来了解诸如 Merkle 树、向量时钟、提示移交、读时修复和其他生僻的名词。



本书针对 Cassandra 0.6 和 0.7 写成。项目组正在努力开发 Cassandra，新的小版本和修订版本会不断释出。在可能的地方，我会尽量解释版本间的不同，不过你在阅读时可能已经用上了一个更新的版本，有些实现因此会有所不同。

更多信息

如果你需要了解关于 Cassandra 的更多信息，获得最近的更新，可以访问本书网站：http://www.cassandraguide.com。

在 Twitter 上关注我（@ebenhewitt）也是个好主意。

格式约定

本书使用了如下排版约定。

- 楷体

  用于标记新名词。

- 等宽字体

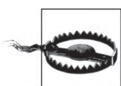
  用于程序代码，在段落中用于表示程序的组成部分，如变量或函数名、数据库、数据类型、环境变量、语句、关键字。

- 等宽粗体
  命令或是其他应该由用户输入的内容。

- 等宽斜体
  应该由用户提供的或由上下文确定的值。



这个图标表明一个提示、建议或一般注记。



这个图标表示一个警告或警示。

使用示例代码

本书用于帮助你完成工作。通常，你可以在程序或文档中使用本书提供的代码。除非你在重新发布我们的大量代码，否则不需要联系我们来获得许可。比如，在程序中使用本书代码的一些片段是无需我们许可的。但是出售或再分发 O'Reilly 的图书示例光盘显然是需要授权的。引用本书或引用示例代码来回答问题是不需要授权的，但使用本书的大量示例代码作为你的产品的文档是需要授权的。

我们乐于见到你在使用时声明引用信息，但不强制要求。引用信息通常包括书名、作者、出版社和 ISBN。比如："*Cassandra: The Definitive Guide* by Eben Hewitt. Copyright 2011 Eben Hewitt, 978-1-449-39041-9."

如果你认为对示例代码的使用需要授权，请通过这个邮箱联系我们 permissions@oreilly.com。

Safari®在线图书



Safari 在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索 7500 多种技术性和创新性参考书以及视频，以便快速地找到需要的答案。

订阅后就可以访问在线图书馆内的所有页面和视频。可以在手机或其他移动设备上阅读。还能在新书上市之前抢先阅读，也能够看到还在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至

打印页面等有用的功能可以节省大量时间。

这本书也在其中。欲访问本书的英文版电子版，或者由 O’Reilly 或其他出版社出版的相关图书，请到 http://my.safaribooksonline.com 免费注册。

我们的联系方式

请把对本书的评论和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到关于本书的相关信息，包括勘误表、示例代码以及其他的信息。本书的网站地址是：

http://www.oreilly.com/catalog/0636920010852/

中文书

http://www.oreilly.com.cn/index.php?func=book & isbn=

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资源中心和网络，请访问以下网站：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

在此，我希望感谢很多帮助我们完成本书的优秀人士。

感谢 Jeremy Hanna 写作了 Hadoop 的章节，也感谢他如此地易于合作。

感谢本书的技术审校者们。特别是 Stu Hood 独到的批注，极大提高了本书的质量。

Robert Schneider 和 Gary Dusbabek 也给出了很有见地的审稿意见。

感谢 Jonathan Ellis，谢谢他为本书作序。

感谢我的编辑 Mike Loukides，旧金山晚餐时优雅的健谈者。

感谢 Rain Fletcher ，他为本书提供了支持，一直鼓励着本书的写作。

我的灵感来自很多了不起的 Cassandra 开发者。向编写出这个优美而强大的数据库的开发者们致敬。

还要一如既往地感谢 Alison Brown，他阅读了手稿、给我提示，并确保我的写作时间，没有他，这本书就不会诞生。

第 1 章

Cassandra 概况

> 如果一个概念一开始不是荒诞不羁的话，那它也没什么前途。
>
> ——阿尔伯特 • 爱因斯坦

欢迎阅读《Cassandra 权威指南》。本书的目标是帮助开发者和数据库管理员们理解这种重要的新型数据库，探索它与传统的关系型数据库系统有何异同，并帮助你在自己的系统中使用 Cassandra。

1.1 关系型数据库有什么问题

> 如果当初我问人们到底想要什么的话，他们会说想要快些的马。
>
> ——亨利 • 福特

请你考虑一种数据模型，它由一个拥有数千名雇员的大公司里的一个小团队发明。这个模型可以通过 TCP/IP 访问，并且可以使用包括 Java 和 Web Service 在内的多种语言访问。在被广泛采纳之前，这个模型起初若非最顶尖的计算机科学家都很难理解，最终因为用的人越来越多，才被大家认识。要使用这种模型构建出数据库，就必须学习许多新术语，换一种方式考虑数据存储。随着相关的产品层出不穷地出现，很多公司和政府部门开始广泛地使用它，因为它非常快——可以在一秒钟内执行上千次操作。这种模型产生了让人难以置信的收益。

然后，又一种新的模型出现了。

这个新模型是一种可怕的异端，主要有两点原因。首先，它最受非议的地方就在于新模型与旧模型完全不同，它们简直是格格不入。这很可怕，因为全新且完全不同的东西通常难以理解。随之而来的争论更进一步巩固了人们的看法，而这些看法主要来自人们已有的工作环境和习得的技术。其次，或许是更重要的原因，新模型所面临的阻碍更多来自商业考虑，它威胁到了在旧模型上的大量投资，这些投资正在带来大量收入。在这个时候改变似乎是可笑的，也是不可能的。

当然，我说的是 1966 年由 IBM 发明的信息管理系统（IMS）分层数据库。

IMS 是为“土星五号”登月火箭而设计的。它的架构师是 Vern Watts，他的整个职业生涯都和 IMS 联系在一起。很多读者都知道 IBM 的 DB2 数据库。而 IBM 广为人知的 DB2 数据库的名字就沿袭自它的前身 DB1，DB1 是围绕着 IMS 的数据模型构建的。IMS 于 1968 年发布，之后在用户信息控制系统（CICS）和其他应用中取得成功。至今 IMS 仍被很多应用使用着。

但是，在 IMS 发明之后的几年里就出现了一个崭新的模型，这个破坏性的、带来威胁的模型就是关系型数据库。

同样来自 IBM 的 Edgar F. Codd 博士在他 1970 年发表的论文《一种用于大规模共享数据存储系统的关系型数据模型》中，阐述了他在 IBM 圣何塞实验室的工作中提出的关系数据模型理论。这篇目前仍然可以在 http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf 访问的论文，成为了后来的关系型数据管理系统的奠基性作品。

Codd 的工作与 IMS 的层次化结构完全对立。IMS 的用户要理解并使用关系型数据库，就需要学习很多听起来十分怪异的新名词。不过，关系型模型与它的前身相比更具优势，部分因为巨人几乎总是站在其他巨人的肩膀上。

关系型数据库这个概念和它的应用已经演化了 40 多年了，毫无疑问，它是软件应用历史上最成功的一个。它既可以在个人小公司里通过微软 Access 数据库来使用，也可以在大型跨国公司的上百台经过调优的服务器上使用，构成存储数 TB 数据的数据仓库。关系数据模型存储了账单、用户记录、产品目录、账户明细、用户鉴权信息等，可以说关系型数据库里差不多存储了整个世界。毫无疑问，无论以现代的技术还是商业视角看，关系型数据库都扮演着关键角色，而且，多年后它也会以各种形式一直伴随我们存在，IMS 也是同样。关系模型虽然是 IMS 的一种替代模型，两者都各得其所。

所以，要问关系型数据库有什么问题，一言以蔽之，就是“没有问题”。

不过，实际上我还想请你考虑一个长一点的答案。这个答案需要从长计议，每个偶

然诞生的概念都在点滴改变着世界，孕育着某种革命。而这一次次的革命不过是历史的重演罢了。从 IMS、RDBMS 到 NoSQL，一如从马到汽车再到飞机，每一个都建立在前一个的基础之上，解决前一个存在的某些问题，每个都有所长，亦有所短。他们彼此共存，直到如今。

那么就来看看为什么现在我们要考虑关系型数据库的替代产品，正如四十年前 Codd 自己面对信息管理系统（IMS）的思考一样——或许 IMS 不是唯一适合用于组织信息、处理数据的方法，可能正是因为某些问题使得他必须要考虑一种 IMS 的替代品。

当基于关系型数据库的应用取得成功，访问量大幅增加的时候，我们就会遇到可扩展性问题。所有具有最基本功能的关系型数据库都会支持 join 操作，不过 join 可能会很慢。由于数据库通常依靠事务来保证一致性，而事务需要锁住数据库的一部分，使之不能被其他用户访问。因为锁本身意味着竞争同一数据的用户会被放入队列，等待获得读写权限，这在高负载的情况下可能会成为系统的死穴。

通常，我们会用下面的一条或几条方法来解决这些问题，很多时候是下面这个顺序。

- 提升硬件能力来解决问题，如增加内存、用更快的处理器以及升级硬盘。这种方法称为*垂直扩展*，可以解一时之忧。
- 当问题再度出现时，解决方法很类似：既然一台机器已经不堪重负了，我们就增加新的计算机，构成数据库集群。不过，这样你就会在正常使用及出故障时遇到数据复制与一致性问题了。这些问题之前从未出现过。
- 现在我们需要更新数据库管理系统的配置。可能是要优化数据库用来写底层文件系统的通道。我们关掉了文件系统的日志，当然，这通常是不推荐的（或者说要具体情况具体分析）。
- 在数据库系统上投入了足够多的精力之后，我们转过来审视自己的应用。我们开始优化索引、优化查询。不过，当我们的应用达到这个规模的时候，恐怕不太会完全没做过索引和查询优化，可能已经优化过不少了。那么，只好重新审视所有数据库访问代码，想发现零星的可以调优的机会，这是一件相当头疼的事。优化内容可能包括减少或改写 join 操作，除去比较消耗资源的特征（如在存储过程中的 XML 处理）等。当然，我们做那些 XML 处理本身肯定是有原因的，如果我们不得不做的话，那就得把它挪到应用层去，希望那里能解决问题，并祈祷我们这么干不会同时把什么其他东西弄坏。
- 我们增加了一个缓存层。对于大系统可能会引入分布式缓存，如 memcached、EHCache、Oracle Coherence 或其他类似产品。现在，我们又需要面临更新缓存和更新数据库的一致性问题了，对于集群来说，问题更加严重。

- 现在我们把注意力重新转向数据库，由于应用已经在那里了，并且我们很了解主要的查询路径，现在我们复制那些访问频率较高的数据，让它们更接近于查询想要得到的形式。这个过程称为反范式化（denormalization），它与关系模型的关键特征里的五种形式是对立的，也违反了 Codd 对关系型数据模型的 12 条建议。我们只能安慰自己说我们是生活在现实世界之中，而非理论世界中，并保证我们所做的都是为了让应用能够在可接受的响应时间下完成工作，即使这已经不是“纯粹”的关系模型了。

我相信这一幕对你肯定非常熟悉。在互联网的规模下，工程师们已经开始考虑这个情况是不是和当年亨利·福特的境遇有些相似，你真正需要的已不仅仅是你本来想的一匹快马了。他们已经做了一些令人称道而有趣的工作。

首先我们必须认识到，关系模型只是一种数据模型而已。它是一种看待世界的有效的方法，对某些问题非常有效。但它并没打算也没被证明可以一统天下，不留任何余地终结所有其他表示数据的方法。如果我们回顾一下历史就会发现，Codd 博士的模型也曾是个破坏性的发明。它是全新的，带来了很多新词汇和像“元组”这样的术语——老词汇但有新意义。关系模型在当时引发了质疑，甚至受到了强烈的批评。即使是 Codd 博士工作的 IBM 公司也是反对者之一，因为他们拥有一系列围绕 IMS 的富有盈利能力的产品，不需要一个闯入者来分蛋糕。

不过如今，关系模型已经无可辩驳地坐上了数据世界中的头把交椅。SQL 获得了广泛的支持和很好的理解，大学的入门课程就会讲授 SQL 与关系数据库。甚至 4.95 美元 1 月就可以租到有免费数据库的互联网主机。我们最终使用什么数据库通常由组织的架构标准而定。即使没有这样的标准，学习一下组织结构已经有了什么数据库平台仍是明智的。我们的开发和架构方面的同事都有相当难得的知识积累。

仅仅是出于渗透或是惯性，我们多年来都已经习惯了关系型数据库是一个四海一家的解决之道。

因此，也许真正的问题不是“关系数据库有什么问题”，而是“你有什么问题要解决”。

也就是说，要确保你的解决方案和你的问题相匹配。关系型数据库确实在解决很多问题时非常有效。

如果你并未面对大规模、弹性扩展的问题，那么对于 Cassandra 之类的复杂系统的取舍完全没有考虑的必要。据我所知，没有任何一位 Cassandra 的支持者会要求别人丢掉他们的所有关系型数据库的知识、放弃他们多年来对于这类系统的难得的经验，或是破坏他们的员工花费数月时间精心构建起来的系统。

关系型数据库长期以来很好地服务于我们这些开发者和 DBA。但是由于互联网，特别是社会化网络的爆炸性发展，相应的必须要处理的数据量也在增长。当 Tim Berner-Lee 在 20 世纪 90 年代初构建 Web 的时候，只是为了物理实验室里的博士们交流科学文档而已。如今，互联网已经无处不在了，从最初的那些科学家到在网上交流小猫图片与表情符号的 5 岁小孩，每个人都在使用互联网。这意味着互联网业必然要支持海量的数据，事实上，这也成为了 Web 巧夺天工的架构的不朽丰碑。

不过，有些基础设施已经开始力不从心了。

在 1966 年，像 IBM 这样的公司可以向世界传播他们的创新。他们自己有问题，也拥有解决问题的能力。但当我们进入 21 世纪第二个十年的时候，我们也开始看到类似的创新了，不过这些创新来自于像 Facebook 和 Twitter 这样的年轻公司。

所以，真正的问题可能也不是“我有什么需要解决的问题”，而是“如果数据不是问题，那么应该怎么来处理这些数据”。如果能够轻易做到容错、跨数据中心可用性、可调整的一致性，以及高达上百 TB 的超大规模、多种可选择的客户端语言，又会怎样？你可能会说，你不需要那种可用性或是那个水平的可扩展性，而且你最清楚你的系统需求。你当然是对的，因为事实上，如果你的数据库不能满足当前对数据库的需求，那么你的系统是根本无法工作的。

我无意通过诡辩来说服你采用一个像 Apache Cassandra 这样的非关系型数据库。我只是想介绍 Cassandra 能做什么、如何做到，这样你就可以依此来决策，并且如果你发现它可用，就可以开始使用它开展实际工作了。只有你自己知道你的数据需求是什么。我不需要你重新考虑自己的数据库，除非你已经被数据库问题折腾得苦不堪言了，或是你需要扩展数据库却无能为力，或是你的数据模型无法足够灵活地匹配你的应用需求。我并不要求你考虑你的数据库，但请考虑你的组织结构，它未来的目标和它将要面临的问题。如果你能收集到有关商业目标的更多数据的话，你会收集么？

不要问如何让 Cassandra 融入你的现有环境，要问你希望解决什么数据问题而不是只关注现在数据存在的问题。要问你希望的新型数据是什么样的。如果可能的话，你想要如何去深入理解你的组织？

1.2 关系型数据库简单回顾

虽然你可能已经很熟悉关系型数据库管理系统（RDBMS）了，我们还是要来关注一下关系型数据库的一些基本概念。这会让我们了解在分布式系统，尤其是那些互联网规模的超大数据系统中有关技术折中考虑的最新理念。

1.2.1 RDBMS：出类拔萃与表现平平

关系型数据库在过去的四十年间一统天下的原因有很多，而其中的一个重要原因便是它有功能丰富、语法简明的结构化查询语言（SQL）。SQL 在 1986 年被接纳为 ANSI 标准，之后有过多个修订版，而且有各个厂商带来的专有扩展语法格式，比如微软的 T-SQL 和 Oracle 的 PL/SQL，都提供了各自附加的针对其实现的特性。

SQL 之所以强大的原因有很多。它允许用户使用语句来表述复杂的数据关系，构成数据处理语言（DML）来插入、选择、更新、删除、截断和合并数据。你可以使用基于关系代数的函数进行丰富的操作，比如查找集合中的最大或最小值，或对结果进行过滤与排序。SQL 语句支持对值进行分组汇聚并执行汇总函数。SQL 也提供了数据定义语言（DDL），可以在运行时直接创建、修改或删除 schema 结构。SQL 还允许你使用同样的语法格式对用户进行授权或取消授权。

另一方面，SQL 非常易于使用。SQL 的基本语法很容易学习，这样就为 SQL 和 RDBMS 降低了门槛。初级 SQL 开发者可以平稳进阶，在软件行业，人们通常面临着快速的变化、紧迫的交付时间和膨胀的预算，在这种压力下，易学易用无疑是非常重要的。SQL 不仅语法很简单，而且还有包括直观的图形界面在内的很多强大的工具可以配合数据库的使用。

部分源于 SQL 是一项 ANSI 标准，所以 RDBMS 可以很方便地和各种系统相集成。所有需要做的工作就是为你的语言编写一个驱动，这样就可以以一种可移植的方式进行对接了。如果你要改变你的应用实现语言（或是换家 RDBMS 厂商），只要你不是太过于依赖数据库厂商的专有扩展，通常都不会太困难。

1．事务、ACID和两阶段提交

除了上面已经提到的特性，RDBMS 和 SQL 还支持事务。根据 Jim Gray 的定义，数据库事务是一种具有 ACID 特征的“状态转移过程”（参考 http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf）。事务的关键特性是它们会首先虚拟地执行，并允许程序员在任务的执行阶段（使用 ROLLBACK）来撤销所有改动。如果一切运行正常，事务就可以可靠提交。对于事务的支持很快就成为了非关系型数据库的讨论中的痛处，所以我们先来看看事务到底意味着什么。

ACID 是原子性（Atomic）、一致性（Consistent）、隔离性（Isolated）和持久性（Durable）的缩写，是验证事务是否正确、是否成功执行的标尺和准绳。

- 原子性

  原子性的含义是“要么全有，要么全无”，也就是说，执行一个语句时，事务中的每个更新都必须成功才能称为成功。不存在有的更新成功，有的更新不成功的部分失败状态。一个最典型的例子就是在 ATM 机上进行汇款服务：汇款需要从一个账号上取钱，再添加到另一个账号里。这个操作是不可分的，两个动作必须全部成功。

- 一致性

  一致性意味着数据必须从一个正确的状态转移到另一个正确的状态，不允许别人看到没有意义的不同的值。比如，如果一个事务试图删除一个客户和她的订单历史，就不应该留下一个订单列指向已经删除的客户的主键，否则，不一致的状态可能会让某些试图读订单记录的操作出错。

- 隔离性

  隔离性是指并发执行的事务不应该彼此依赖，它们运行在各自的空间之中。也就是说，如果有两个事务同时需要修改同一份数据，那么其中之一就必须等待另一个完成之后再操作。

- 持久性

  一旦一个事务成功完成，变更就不再丢失。这并不意味着其他事务稍后不可以修改同一数据，而是说写数据的程序可以确信变更在下一个事务运行前已经就绪。

表面上看，这些属性似乎是显而易见的，甚至不值得讨论。估计所有运行数据库的人都相信，更新数据必须要花费一定时间，因为执行更新的关键点就在于，这些数据也是要被其他人读取的。然而，进一步的思考可能让我们希望找个方法来调整一点这些属性，略微控制它们一下。所谓“互联网上没有免费的午餐”，一旦发现究竟要为事务付出多少代价，我们可能就会开始考虑，是否需要个替代方案了。

事务在高负载下变得非常困难。当你开始尝试水平扩展一个关系型数据库、让它分布化的时候，就必须要考虑*分布式事务*的问题了。这里，事务不再是一张表或一个库里的简单操作了，而是一个跨越多个系统的操作。为了继续保持事务 ACID 属性的“荣誉”，你需要一个精巧设计的跨节点的事务管理器。

为了保证事务跨越多主机成功执行，人们提出了两阶段提交（有时被称为 2PC，即 two-phase commit）。但是因为两阶段提交锁住了所有的相关资源，这种方法只对可以很快完成的操作可用。虽然很多时候分布式操作都可以在不足一秒的时间内完成，但情况并非总是如此。在有些需要跨越多台主机的情况下，你无法准确控制自己的用时，有些需要协调多个相关动作的操作甚至可能会耗时数小时。

两阶段提交是*阻塞式*的，也就是说客户端（竞争消费者）必须等待前一个事务完成，否则就无法访问阻塞了的资源。两阶段提交需要等待一个节点的响应，但或许这个节点已经死了。当然，可以设置超时来允许事务协调节点发现某个节点已经没有响应了，我们可以避免一直等下去。但是，两阶段提交仍然可能会导致死循环，因为一个节点可以发送一个消息给事务协调节点，声明自己已经完成任务，可以提交了，然后它就会等待协调节点发回提交响应（或者如果节点不能提交，就等待回退响应），但如果这时协调节点宕机了，那么这个不幸的节点就得永远等下去了。

为了解决两阶段提交的分布式事务的问题，数据库领域中有人提出了*补偿*的概念。补偿在 Web 服务中非常常用，简单地说就是把操作立刻提交，如果发现了什么错误的话，就调用一个新的操作来恢复开始的状态。

架构师们经常要用一些基本的、众所周知的补偿模式作为两阶段提交的替代方案，包括失败后核销事务、放弃出错事务以及事后核对等。另一个替代两阶段提交的方法是得到通知时重试出错操作。对于预定系统或是股票销售记录系统，这些方法可能无法满足需求。但对于其他应用，比如计费或是售票应用，这是可以接受的。



Google 的架构师 Gregor Hohpe 曾经写过一篇题为“星巴克不使用两阶段提交”的非常精彩、被广泛引用的博客。文中介绍了现实世界中的两阶段提交是多么难以扩展，并特别介绍了一些替代方法。这篇文章浅显、有趣而富有启示，值得一读。链接地址是：http://www.eaipatterns.com/ramblings/18_starbucks.html。

2PC 给应用开发者引入的问题包括可用性的损失和部分出错时导致的高时延，两者都是难以接受的。所以一旦你的事业发展顺利，需要扩展过去的单机数据库的时候，你必须要弄明白如何在保持 ACID 属性的同时处理好跨越多台机器的事务问题。不论你是有 10 台、100 台还是 1000 台数据库服务器，事务都和运行在一个节点上一样需要具备原子性。只是这下子任务要艰巨得多了。

2．schema

关系型数据库的一个经常被称赞的特性就是它们提供了丰富的 schema。通过 schema，你可以使用关系模型表述各种应用领域中的对象。在业界有很多高级（又昂贵）的工具，如 CA 的 ERWin Data Modeler 可以帮你做到这点。但是，为了创建正确、规范的 schema，你不得不创建某些表，来容纳一些在应用领域的业务中并不存在的对象。比如，一个大学的数据库里可能包含了学生表和课程表。但因为这是一个多对多的关系（一个学生可以同时修多门课程，而一个课程同时会有多名学生上），你不得不创建一个联合表。这污染了一个朴素的数据模型——我们所需要的其

实只是学生和课程。这也强迫我们不得不写更复杂的 SQL 语句来将这些表联合到一起使用。而且 join 语句可能会很慢。

对于不算大的系统，这根本不是问题。但是当你需要在多张表里同时处理很多行的时候，复杂的查询和 join 操作可能会慢得难以承受。

最后，并不是所有的 schema 都很好地符合关系模型。在过去 10 年中，复杂事件处理系统的应用非常广泛，它使用一个非常快速的流来表示状态变化。这种系统常常用于将运行时的相关事件与其他可能相关的事件进行对比，来分析得到结论以支持商业决策。虽然事件流可以在关系型数据库中描述，但这种表达方式的扩展却十分别扭。

如果你是一个应用开发者，应该对很多对象关系映射（ORM）框架非常熟悉，近年来很多此类框架可以简化应用对象到关系模型的映射难度。同样，对于小系统，ORM 非常简单。但 ORM 同样引入了新的问题，比如内存需求大，而且，ORM 框架引入的大量笨拙的映射代码也污染了应用的代码。下面是一个在 Java 中使用 Hibernate 来“减轻”编写 SQL 代码的负担的例子：

```
@CollectionOfElements
@JoinTable(name="store_description",
    joinColumns = @JoinColumn(name="store_code"))
@MapKey(columns={@Column(name="for_store",length=3)})
@Column(name="description")
private Map<String, String> getMap() {
    return this.map;
}
//...
```

是不是我们根本没有解决这里的问题？当然，尤其对于那些与服务和基于 XML 的应用有大量文档交换的系统，并不总能简洁地映射关系型数据库。这更会加剧这个问题。

3．分片与shared-nothing架构

> 如果你无法拆分系统，就无法扩展它。
>
> ——Randy Shoup，eBay 杰出架构师

另一种用于扩展关系型数据库的方法是在系统架构中引入分片（sharding）。这种方法已经在很多大型网站和 Web2.0 应用中取得了成功，比如 eBay，分片架构可以支持每天数十亿次的 SQL 查询。分片的思路就是将数据拆分，不把所有数据放到一个单独的服务器上，也不把数据复制到集群中的所有服务器上，而是把数据水平地进行分割，并分别存放。

例如，一个关系型数据库里有一个很大的顾客表。（从程序的角度看）最省事的扩容方法就是通过增加 CPU、内存，更换更好的硬盘来进行垂直扩展了，可是一旦你的

事业继续成功，获得了更多的用户，在某一时刻（可能是达到上千万行的时候），你可能不得不开始考虑无法增加更多机器的问题了。再加机器的话，是否要向所有机器复制所有数据？或者是否把单张顾客表拆开，每个数据库只有一部分记录和相关的订单信息？这样，当客户进行查询的时候，负载将只压在拥有相关记录的机器上，而对其他机器不产生影响。

显然，要进行分片，你必须找到一个合理的键值来对记录进行划分。比如，你可以按照首字母来进行划分，把顾客记录分布到 26 台机器上，每台机器仅存储姓氏以一个字母开头的顾客的记录。不过这似乎不是一个好的策略，很少有人的姓氏以字母 Q 或 Z 开头，这两台机器会一直很闲，而存放 J、M 和 S 开头姓氏的顾客的机器会一直很忙。你还可以以某些数字来进行分片，比如电话号码、成为会员的日期，或是顾客所在州的名字。如何分片完全取决于你希望数据如何分布。

这里有三种基本的确定分片结构的策略。

- 按照特性或功能来分片
  这是 eBay 的杰出架构师 Randy Shoup 采用的方法，他曾在 2006 年帮助 eBay 建立了成熟的架构，可以支持每天数十亿次查询。使用这个策略，并不是将一个表中的记录进行拆分（比如之前提到的顾客表），而是分成多个功能上互相不怎么重叠的数据库。比如，在 eBay，用户分到一个分片中，而出售的产品则放到另一个分片中。在 Flixster，电影评分放到一个分片中，而评论则放到另一个分片里。这个方法需要深入理解应用领域，才能更好地分片。

- 基于键值的分片
  这种方法需要在你的数据中找到一个可以均匀分布到各个分片中的键值。不像在刚才的例子中那样，非常幼稚地通过字母表的方式简单进行分布，而应当对一个关键数据元素进行一次单向哈希，根据哈希值将数据分布到多台机器上。这种策略经常会使用一个时间或是数值键进行哈希。

- 查表法
  在这种方法里，集群中的一个节点会充当黄页目录的角色，用于查询哪个节点拥有用户要访问的数据。这种方法有两个显而易见的问题：首先，由于每次查询都需要进行查表操作，这会影响到查询的性能；其次，这个表不仅会成为一个瓶颈，还可能是系统中的一个单点失效点。



http://lsvp.wordpress.com/2008/06/20 介绍了 Flixster 是如何使用分片策略来改进网站性能的。

依据不同的分片策略，可以减小资源竞争，不仅允许你水平扩展系统，而且可以更精确地扩展，因为你可以精确地提升特定分片的性能。

分片可以看做是一种 shared-nothing 的数据库架构。所谓 shared-nothing 架构是指分布式系统中，不存在集中存储的（共享）状态，也就无需竞争共享资源了。这一概念是由加州大学伯克利分校的 Michael Stonebraker 于 1986 年在他的论文“The Case for Shared Nothing”中提出的。

shared nothing 近来在 Google 被发扬光大了，他们实现了不要共享状态的 BigTable 数据库和 MapReduce 分布式计算框架，这些系统可以近乎无限地扩展。Cassandra 数据库也是 shared-nothing 架构的，没有中心控制节点，也没有主从之分，所有节点是对等的。



你可以在 http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf 阅读这篇 1986 年的论文“The Case for Shared Nothing”。这篇论文非常简短，只有寥寥数页。如果你看看的话，就会发现很多 shared-nothing 分布式系统架构的特征，诸如易于达到高可用性和易于扩展到大量节点等都恰恰是 Cassandra 所擅长的。

MongoDB 还提供了自动分片能力，用于失效备援和负载均衡。很多非关系型数据库都提供了这项自动功能，而且用起来非常方便。手工创建和维护自定义的分片是个技术活。从数据架构的宏观层面了解分片十分有必要，但具体到 Cassandra，它是自动化的，使用了一种类似基于键值分片的方法，将数据分布到不同节点上。

4．小结

总之，关系型数据库非常擅长于解决某些数据存储问题，但因为其自身特点，当系统需要扩展的时候，会受到很强的制约。扩展时，你常常得想方设法避免 Join 操作，这意味着数据的反范式化，意味着需要保存数据的多个副本，以及严重地破坏了你对数据库和应用的最初设计。而且，你几乎必然要考虑分布式事务的问题，这很快就将成为一个瓶颈。除了最昂贵的 RDBMS，其他的都不支持补偿操作。而即使你花费巨资买了最昂贵的数据库，你仍然需要对那些无法忽视分布式事务的地方十分谨慎地选择分片所使用的键值。

或许，更重要的是，随着我们讨论 RDBMS 的限制，和随之产生的用于克服扩展性问题所使用的架构策略，一幅巨图正在慢慢展开。可以看出，一些 NoSQL 技术或许与我们的最初设想相比其实并不激进，也不算吓人，倒更像是一些我们管理大数据库时就在做的工作的自然表达和封装。

1.2.2 互联网的规模

> 一个发明必须对它完成时的世界有意义，而非它开始时的。
>
> ——Ray Kurzweil

基于 RDBMS 的一些内在的设计初衷，它并不像其他的系统一样那么容易进行扩展，尤其是相比于一些新近的，在设计时考虑了互联网的结构的系统。不过，我们不但需要考虑互联网的静态结构，还需要考虑它难以置信的扩张速度，因为随着越来越多的数据积累，我们希望系统架构可以让我们的组织利用即时的数据来支持决策，为用户提供更强大的新功能和高性能。



有一个不太容易考证的传说，17 世纪的英国诗人 John Milton 阅读过地球上曾经出版过的所有书。Milton 精通多种语言（他去世前甚至还在学习印第安部落纳瓦霍的语言），而且那个年代每年出版的图书大概是上千册，所以，全都读过也并非不可能。但从那时至今，数据的增量就难以记述了。

众所周知，互联网还在快速发展中。现在让我们花一点时间来看看 IDC 的研究报告《扩展中的数字世界》中的一些数字。（全文可以在 http://www.emc.com/collateral/analyst-reports/ expanding-digital-idc-white-paper.pdf 获得。）

- YouTube 每天播出 1 亿段视频。
- Chevron 每天积累 2 TB 数据。
- 2006 年，互联网的总数据量大约是 166 EB，而到了 2010 年，这个数字大约是 1000 EB 了。1 EB 是 10^{18} 字节，即 110 万 TB。更直观地说，1 EB 大约相当于 5 万年长的 DVD 画质的视频，而 166 EB 大约是所有图书的信息量总和的 300 万倍。
- 沃尔玛的顾客交易数据库在 2000 年大约是 110 TB，每天记录大约上千万条。到了 2004 年，这个数据库已经有大约 5000 TB 了。
- 电影阿凡达的制作过程中大约需要用到 1 PB 的存储空间，如果这是 1 首 MP3 的话，可以播放 32 年（来源：http://bit.ly/736XCz）。
- 2010 年 5 月，Google 每天发售 10 万部 Android 手机，所有这些手机都以互联网访问为基本服务。
- 在 1998 年，全球 email 账号大约有 2.53 亿个，到 2010 年，这一数字接近 20 亿。

可见，互联网上需要存储、处理与查询各种数据，使用这些数据的业务也各不相同，不仅需要考虑零售供应商们的客户数据，也不仅是数字视频，还有需要数字化的电视节目、爆炸式增长的 email、即时消息、手机、射频识别（RFID）、IP 电话（VoIP），等等。现在我们有蓝光播放器播放音频和视频流。随着我们开始离开物理存储介质，提供媒体内容的公司，以及在他们外围提供增值服务的第三方，都需要

极为可扩展的存储解决方案。再考虑一下，作为典型的应用开发者或数据库管理员，我们可能习惯于把关系型数据库看做是世界的中心。但其实在企业中，80% 的数据都是非结构化的。

你或许觉得 Cassandra 之类的 NoSQL 解决方案所提供的大规模与你无关。可能确实如此，可能你并没有遇到 Cassandra 能帮得上你的问题。我也并没有让你直接对现有的数据库和里面的数据动手，将它们迁移到 Cassandra。那将是非常困难的一项工作，而且短时间内很难收到回报。几乎可以肯定，你现有的数据库是最适合你的应用系统的。但如果能够并入更多更丰富的数据集，来改进你的应用，你希望数据库可以有什么样的品质呢？这个问题进一步转化为，如果有一个可靠、有弹性、可扩展、超大容量而且写入速度超快的存储系统，你希望你的应用变成什么样子呢？

立足于当今互联网的规模，面向未来，Apache Cassandra 可能是你的答案之一。

1.3 Cassandra的电梯间演讲

不论是好莱坞的剧作家还是初创的软件公司，都被建议准备好他们的“电梯间演讲”。这个演讲是他们的产品的简洁介绍，要求是在一两分钟内简明扼要地介绍他们的产品，以便在万一能很幸运地和一个主管、代理或是投资人一起挤上同一个电梯的时候，能够说服他来投资这个项目。Cassandra 如此引人入胜，让我们来浓缩一个电梯间演说，以便你能说服你的经理和同事给 Cassandra 一个机会。

1.3.1 50个字介绍Cassandra

“Apache Cassandra 是一个开源的、分布式、无中心、弹性可扩展、高可用、容错、一致性可调、面向列的数据库，它基于 Amazon Dynamo 的分布式设计和 Google BigTable 的数据模型，由 Facebook 创建，已经在一些最流行的网站中取得了应用。”英文刚好 50 个字，中文也不到 100 个。

当然，如果你只是在电梯里背这些说词给你的老板的话，可能只能得到一个白眼。所以，让我们在后面的小节里分别讨论一下这些关键点。

1.3.2 分布式与无中心

Cassandra 是分布式的，这意味着它可以运行在多台机器上，并呈现给用户一个一致的整体。事实上，在一个节点上运行 Cassandra 毫无意义。虽然你确实可以这么做，而且这可以帮你了解它的工作机制，但你很快会意识到，需要多个节点才能真正了解 Cassandra 的好处。它的很多设计和实现让系统不仅可以在多节点上运行，更为

多机架部署进行了优化，甚至一个 Cassandra 集群可以运行在分散于各地的多个数据中心上。你可以放心地将数据写到集群中的任意一台机器上，Cassandra 都会收到数据。

对于很多存储系统（如 MySQL，Bigtable），一旦你开始扩展它，就需要把某些节点设为主节点，其他则作为从节点。但 Cassandra 是无中心的，也就是说每个节点都是一样的，没有节点会承担特殊的管理任务。与主从结构相反，Cassandra 的协议是 P2P 的，并使用 gossip 来维护存活或死亡节点的列表。

*无中心*这一事实意味着 Cassandra 不会存在单点失效。Cassandra 集群中的所有节点的功能都完全一样，有时这被叫做“服务器对称”。与你见过的 MySQL、BigTable 这样的主从式的系统不同，因为 Cassandra 的所有节点的工作都是相同的，从根本上就没有一个特殊的主机作为主节点来承担协调任务。

在很多分布式存储方案中（比如 RDBMS 集群），你需要在一个叫做 replication 的进程里设置，让数据在不同服务器上存放多个副本，它会把数据复制到多台机器上，这样这些机器就可以同时提供服务，从而提高系统性能。通常这个进程不采用 Cassandra 这样的无中心设计，而是依照预先设定的*主从角色*来工作。这样，这种集群中的服务器的功能就不是相同的。需要让集群中的一个服务器作为主节点，而其他则作为从节点。主节点是最终数据控制者，与其他节点间是单向的数据同步关系。如果主节点坏了，整个数据库就难以为继了。所以，无中心的设计是 Cassandra 的高可用性的关键所在。注意，虽然我们经常在 RDBMS 里看到主从副本机制，但很多 NoSQL 数据库也是主从结构的，MongoDB 就是一例。

总之，无中心架构有两个关键优势：不仅比主从架构更简单，而且可以避免系统宕机。比起主从结构的数据存储系统来说，无中心架构的维护更方便，因为所有节点都可以一视同仁。这也就意味着你不必为扩展来考虑更多的东西，搭建一个 50 个节点的系统和搭建一个单节点的系统没什么区别。也没有什么配置上的特别需求来支持节点扩展。更进一步，对于主从结构来说，主节点很容易出现单点失效（SPOF）。要想避免单点失效，你常常需要增加系统的复杂度，构成多主节点。而因为所有 Cassandra 的节点都是一样的，损失任何一个节点对系统服务都没什么大影响。

综上所述，因为 Cassandra 是分布式、无中心的，它不会有单点失效，所以支持高可用性。

1.3.3　弹性可扩展

可扩展性是指系统架构可以让系统提供更多的服务而不降低使用性能的特性。仅仅

通过给现有的机器增加硬件的容量、内存进行垂直扩展，是最简单的达到可扩展性的手段。而水平扩展则需要增加更多机器，每台机器提供全部或部分数据，这样所有主机都不必负担全部业务请求。但软件自己需要有内部机制来保证集群中节点间的数据同步。

*弹性可扩展*是指水平扩展的特性，意即你的集群可以不间断服务地扩展或缩减规模。要做到这点，集群必须能够在不大幅修改或重新配置的情况下，接收那些新加入、获得全部或部分数据的节点，让它们开始提供服务。你不需要重新启动进程，不必修改应用的查询，也无需自己手工重新均衡数据分布。在 Cassandra 里，你只要加入新的计算机，Cassandra 就会自动地发现它并开始让它工作。

当然，缩减规模意味着你要从集群中移走部分处理能力。当你的应用要迁移到别的平台上，或是应用失去用户，你不得不卖掉部分硬件的时候，你可能不得不这么做。让我们祈祷这永远都不会发生吧。不过万一要是发生了，你也不需要为规模的缩减来让整个系统变得混乱不堪。

1.3.4 高可用与容错

从一般架构的角度看，系统的可用性是由满足请求的能力来量度的。但计算机可能会有各种各样的故障，从硬件器件故障到网络中断都有可能。任何计算机都可能发生这些情况。当然有某些非常复杂（通常也特别昂贵）的计算机可以自己应付很多情况，它们一般都有硬件冗余，并在发生故障事件的情况下会自动响应并进行热切换。但任何人都可能会偶然碰断以太网线，这时一个单独的数据中心就会有灾难发生，这些都是无法避免的。所以，对于一个需要高可用的系统，它必须由多台联网的计算机构成，并且运行于其上的软件也必须能够在集群条件下工作，有设备能够识别节点故障，并将发生故障的中断的功能在剩余系统上进行恢复。

Cassandra 就是高可用的。你可以在不中断系统的情况下替换故障节点，还可以把数据分布到多个数据中心里，从而提供更好的本地访问性能，并且在某一数据中心发生火灾、洪水等不可抗灾难的时候防止系统彻底瘫痪。

1.3.5 可调节的一致性

*一致性*的基本含义是读操作一定会返回最新写入的结果。考虑电子商务网站的两个客户同时要把一个商品放到他们的购物车里的情景。如果我在你刚刚拿走最后一个商品的时候也要把商品放到自己的购物车里，那么你应该得到这个商品，而我应该被告知商品已经售完了。在把状态一致地写到所有有此数据的节点的时候，这点是得到保证的。

但是，天下没有免费的午餐，后面我们还会看到，扩展数据存储系统就意味着我们不得不在数据一致性、节点可用性和分区耐受性之间做某些折衷。Cassandra 常被称作是“最终一致性”，这实际有点让人误解。简单地说，Cassandra 牺牲了一点一致性来换取了完全的可用性。但是 Cassandra 实际更应该表述为“可调一致性”，它允许你来方便地选定需要的一致性水平与可用性水平，在二者间找到平衡点。

“最终一致性”这个名词让业界有些骚动，一些人对于一个被称为“最终一致”的系统非常不放心，这里让我们来具体解读一下。

最终一致性的批评者有一个比较宽容的说法。或许最终一致性对社交应用可能还可以，因为数据并不*那么*重要。毕竟你给妈妈贴出了小 Billy 吃了什么早餐这样的事情即使丢了也没什么大不了的。但*我*的数据却非常重要，在我的数据模型里，绝对不会允许最终一致性这种可笑的事情发生。

可是事实上，大部分最流行的网络应用（亚马逊、Facebook、Google、Twitter）都在使用这一模型，它应该确实有可取之处。这些数据对于运营这些应用的公司来说也应该是非常重要的，毕竟这是他们赖以生存的产品，而且这些公司在竞争如此激烈的世界里已经成长为数十亿美元营收的巨头，拥有几十亿用户。或许真的有系统可以保障在这样一个高流量、接入不同网络的系统里，能够得到即时、有保障而且完美的一致性，不过你现在可能还找不到这样的完美系统。

批评者们抱怨 Cassandra 之类的海量数据库只支持最终一致性，而其他分布式系统可以支持*严格*一致性。但是世界如此丰富多彩，事实也从来不是非黑即白，要么一致、要么不一致这样的二元论并不能真实地反映实际情况。实际上一致性也是有很多不同*级别*的，而且在真实世界里，一致性也会受到不同外部环境的极大影响。

最终一致性是架构师们可选的多种一致性模型中的一个。让我们现在来看看这些做了不同折中的模型。

- 严格一致性
  有时也叫做顺序一致性，是最严格的一致性要求。它要求所有读操作总是返回最新的写结果。这听起来很好，似乎确实是我们所需要的。就选它好了！但是，再仔细看看，我们还会看到什么？“最新的写结果”到底意味着什么？最新写给谁的？在一个单处理器的机器里，不会看出任何问题，因为无论如何操作都会逐一顺序进行。但是，当系统分布在位置分散的多个数据中心的时候就变得不那么可靠。要达到严格一致性，需要某种全局锁机制来给每个操作加上一个时间戳，不论数据位于何处，用户位于何处，也不论得到响应需要访问多少服务，而且这些服务还可能是分散的。

- 因果一致性
  这种一致性模型比严格一致性稍弱。这种模型消除了幻想中的需要同步一切操作的单一的全局锁，避免了无法承受的瓶颈。因果一致性不依赖于时间戳，它使用更为语义化的方法，尝试去判断事件的原因，并按照因果关系来达到一致性。这意味着潜在相关的写操作必须被顺序读出。如果两个彼此无关的不同操作同时写同一个域，那么这些写操作可以被推断出并不是因果相关的。而如果一个操作紧接着一个进行，这可能就是因果相关的了。因果一致性要求，相关的写操作必须被顺序读出。

- 弱一致性（最终一致性）
  最终一致性从字面上解释就是更新终将传播到整个分布式系统的每个角落，但这需要一定的时间。最终，所有副本都会是一致的。

当你考虑需要如何才能达到更好的一致性的时候，最终一致性忽然变得很有吸引力。

在考虑一致性、可用性和分区耐受性时，对于一个给定的系统，我们只能达到其中的两个目标（我们会在下一节 Brewer 的 CAP 理论来详细讨论这个问题）。中心问题实际是多副本数据的更新问题。要达到强一致性，所有副本上的操作都必须同步进行，这就意味着它们必须要阻塞，锁定所有的副本，直到操作完成，这一过程中，系统必须被阻塞，锁定所有的副本，所有有竞争关系的客户端都不得不等待这一操作的完成。这个系统的副作用就是，一旦系统中的某些节点出故障的时候，有些数据就成为不可用的了。正如亚马逊的 CTO Werner Vogels 所说的："数据在被确定完全正确之前都是不可用的，而不用去试图解决答案的正确性。"（参考"Dynamo: Amazon's Highly Distributed Key-Value Store"，http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html，第 207 页。）

我们可以换用更乐观的方法来进行副本复制——为了避免影响客户端的操作，在后台进行更新的传播工作。这种方法的难点在于，现在我们又遇到如何判断和解决冲突这个问题了。在设计一个系统时，我们必须判断，是在读的时候解决冲突还是在写的时候解决冲突。对于一个分布式数据库的设计者，必须要进行一个抉择——数据库应该总是可读的还是总是可写的。

Dynamo 和 Cassandra 的选择是总是可写，而把判断冲突的问题留给了读操作，从而获得了很好的性能增益。另一种方案可能会在网络或服务器发生故障的时候拒绝更新操作。

在 Cassandra 中，一致性并非一个"要么全有，要么全无"问题，我们或许应该更精确地称之为"可调的一致性"，因为客户端可以控制在更新到达多少个副本之前，必须阻塞系统。这是通过设置副本因子（replication factor）来调节与之相对的一致性级别。

通过*副本因子*（replication factor），你可以决定准备牺牲多少性能来换取一致性。副本因子是你要求更新在集群中传播到的节点数（注意，更新包括所有增加、删除和更新操作）。

客户端每次操作还必须设置一个*一致性级别*（consistency level）参数，这个参数决定了多少个副本写入成功才可以认定写操作是成功的，或者读取过程中读到多少个副本正确就可以认定是读成功的。这里，Cassandra 把决定一致性程度的权利留给了客户自己。

所以，如果需要的话，你可以设定一致性级别和副本因子相等，从而达到一个较高的一致性水平，不过这样就必须付出同步阻塞操作的代价，只有所有节点都被更新完成才能成功返回一次更新。而实际上，Cassandra 一般都不会这么来用，原因显而易见（这样就丧失了可用性目标，影响性能，而且这不是你选择 Cassandra 的初衷）。而如果一个客户端设置一致性级别低于副本因子的话，即使有节点宕机了，仍然可以写成功。

1.3.6 Brewer的CAP理论

要理解 Cassandra 的设计和它所谓的"最终一致性"数据库，我们首先需要了解一下 CAP 理论。CAP 理论由 Eric Brewer 提出，所以有时又被称为 Brewer 理论。

2000 年，当时在加州大学伯克利分校工作的 Eric Brewer 在 ACM 分布式计算原理会议上提出了 CAP 理论。依据这个理论，在一个大规模分布式数据系统中，有三个需求是彼此循环依赖的：一致性、可用性和分区耐受性。

- 一致性（Consistency）
  对于所有的数据库客户端使用同样的查询都可以得到同样的结果，即使是有并发更新的时候也是如此。

- 可用性（Availability）
  所有的数据库客户端总是可以读写数据。

- 分区耐受性（Partition Tolerance）
  数据库可以分散到多台机器上，即使发生网络故障，被分成多个分区，依然可以提供服务。

Brewer 理论是说，对于任意给定系统，只能强化这三个特性中的两个。这很类似于软件开发中的名言："你可以让软件很好，让它很快，或者很便宜：不过三个里面你只能选择两个。"

由于循环依赖关系，我们不得不在它们之中做出选择。比如，你期望获得更强的一致性，那可能就只能拥有较低的分区耐受性，除非你在可用性上做一些让步。

CAP 理论在 2002 年被 MIT 的 Seth Gilbert 和 Nancy Lynch 所证明。不过，在分布式系统中你将不得不面对网络分区的问题，而且在某些时候，机器也常常出现故障，从而导致某些节点不可达。丢包同样是与生俱来的问题。所以，我们可以得到结论，一个分布式系统必须尽力在网络发生分裂的情况下继续工作（具有分区耐受性），这样我们实际上只能在剩下的两个特性里二选一：可用性或是一致性。

如图 1-1 所示，三个特性没有相互交叠的区域。

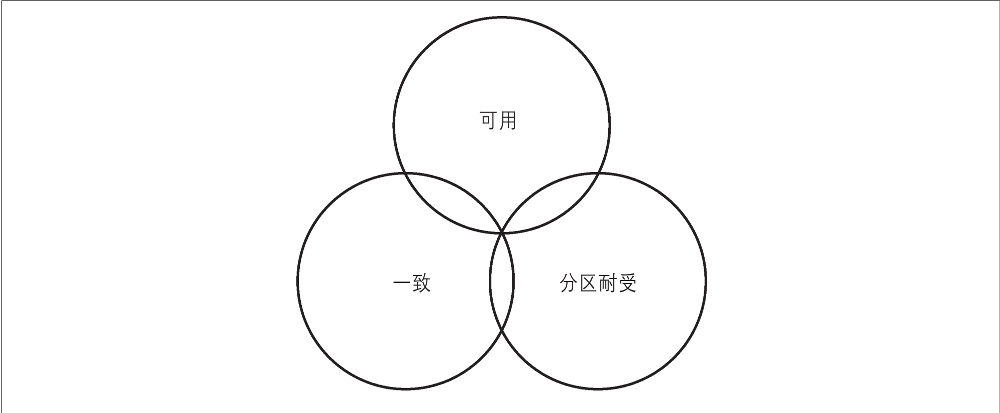


图 1-1：CAP 理论指出，同时只能具有这三个特性中的两个

通过图示来观看每个不同的非关系型数据存储系统在 CAP 图谱中排在什么位置，可能更有助于理解这个概念。图 1-2 是在 2009 年 MongoDB 的创始人和 CEO，Dwight Merriman 在纽约 MySQL 用户组演讲的幻灯片的基础上绘制的（你可以在 http://bit.ly/7r6kRg 查看这个幻灯片）。这里，我根据我的研究修改了某些系统在图中的位置。

图 1-2 显示了我们本章中讨论的几种不同数据库系统的关注焦点。注意，图中的数据库的位置与其具体配置有关。正如 Stu Hood 指出的，一个分布式 MySQL 数据库只有在使用了 Google 的同步复制补丁的时候才可以被认为是一致的系统，否则，它应该只能被看做是可用和分区耐受的系统。

非常有趣的一点是，一个系统在 CAP 体系中所处的位置和一个存储机制最初被发明时想要解决的问题，两者并不一定是一致的，比如，CP 边上既有图数据库，也有面向文档的数据库。

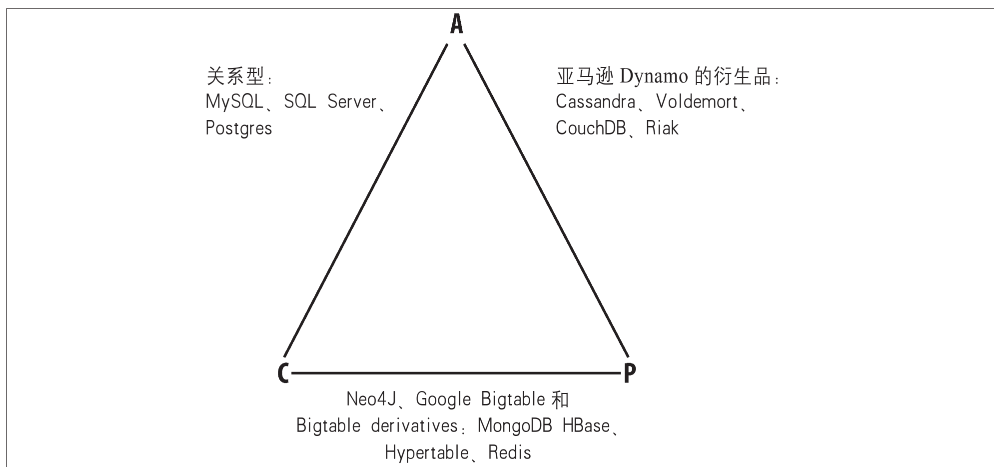


图 1-2：数据库在 CAP 连线上的位置

在这张图里，关系型数据库通常会在一致性和可用性的连线上，这意味着它们在网络失效的时候（比如网线中断）都会无法工作。这通常是由于设置了一个单主节点造成的，因为主节点可能发生故障，其他的一组服务器也可能没有足够的机制让它们在网络分裂的情况下能继续提供服务。

诸如 Neo4J 之类的基于图的数据库和或多或少沿袭自 Google 的 BigTable 的设计的数据库（如 MongoDB、HBase、Hypertable 以及 Redis）全都较少地关注可用性，更多地强调一致性和分区耐受性。



如果你对海量数据存储或是 NoSQL 数据库的特性有兴趣，可以看一下本书附录。

最后，包括 Cassandra、Voldemort、CouchDB 和 Riak 在内的沿袭自 Amazon 的 Dynamo 的设计的数据库都更强调可用性和分区耐受性。不过，这并不意味着它们认为一致性不重要，它们舍弃的一致性并不比 Bigtable 舍弃的可用性更多。按照 Bigtable 的论文，“某些”数据的平均不可用时间大约为 0.0047%（论文第 4 节），这就是说，这些特性都是相对的，因为我们是在讨论一些已经非常可靠的系统。你可以把（C，A，P）三个字母看做是几个旋钮，按照你自己的需求来调节系统，Dynamo 类的系统倾向于那些可以容忍“最终一致性”的场合，这里的“最终”实际也就是毫秒级的时间，读时修复意味着读操作可以返回一致的结果，而且如果需要，你也可以达到强一致性。

那么，实际上，在 CAP 中只能三选二到底意味着什么呢？

- CA

  主要支持一致性和可用性，这意味着你很可能使用了两阶段提交的分布式事务。也就是说，如果网络发生分裂，那么系统可能会停止响应，这也意味着你的系统很可能被限制在一个数据中心集群以降低网络分区发生的可能性。如果你只需要这个级别的规模扩展，那么可以选择 CA 取向的系统，它较易于管理，允许你使用简单而且熟悉的结构。

- CP

  主要支持一致性和分区耐受性，你可以通过改进系统架构，设置数据分片来提升可扩展性。你的数据将保持一致性，但如果有节点发生故障，仍然会有部分数据无法访问（不可用）。

- AP

  主要支持可用性和分区耐受性，你的系统可能返回不太精确的数据，但系统将始终可用，即使是网络发生分区的时候也是如此。DNS 可能是这类系统中最为著名的例子了，这类系统可扩展性非常强，高可用，而且具有分区耐受性。



注意，这里的介绍是要给出一个概观，以便从比较高的维度来对这些系统进行比较，但实际系统的区分并不是这么绝对。比如，Google 的 BigTable 在这个体系中究竟应该放在什么位置就并不确切。Google 的文件中声称 BigTable 是“高可用的”，不过在文件后面的部分却说，如果 Chubby（BigTable 的持久化锁服务）由于服务故障或是网络问题持续不可用超过一段时间的话，BigTable 将变得“不可用”（论文第 4 节）。对于读操作，文件说，“我们没有考虑数据的多个副本和其他由于视图或是索引造成的不同形式的多副本的可能性。”最后，文件指出“BigTable 不想解决中心控制和拜占庭容错”（第 10 节）。由于这些不太一致的信息，你可以发现确定一个数据库的 CAP 到底在什么样的水平并不是一门精确科学。

1.3.7 面向行

Cassandra 经常被看做是一种“面向列”的数据库，这也并不算错。它的数据结构不是关系型的，而是一个多维稀疏哈希表。“稀疏”意味着任何一行都可能会有一列或者几列，但每行都不一定（像关系模型那样）和其他行有一样的列。每行都有一个唯一的键值，用于进行数据访问。所以，虽然说 Cassandra 是面向列的数据库不是什么错，但更确切地说，应该把 Cassandra 看做是一个有索引的、面向行的存储系统，第 3 章会更详细地解释这个问题。我把面向数据当做是 Cassandra 的一个特征，

因为在非关系型模型里，有多种易于可视化和使用的数据模型，而且，不考虑具体的应用就把关系型模型看做是最佳解决方案的结论下得也为时尚早。

Cassandra 的数据存储结构基本可以看做是一个多维哈希表。这意味着你不必事先精确地决定你的具体数据结构或是你的记录包含哪些具体字段。这特别适合处于草创阶段，还在不断增加或修改服务特性的应用。而且也特别适合应用在敏捷开发项目中，不必进行长达数月的预先分析。对于使用 Cassandra 的应用，如果业务发生变化了，只需要在运行中增加或删除某些字段就行了，不会造成服务中断。

当然，这不是说你不需要考虑数据。相反，Cassandra 需要你换个角度看数据。在 RDBMS 里，你得首先设计一个完整的数据模型，然后考虑查询方式，而在 Cassandra 里，你可以首先思考如何查询数据，然后提供这些数据就可以了。

1.3.8 无schema

你需要定义 Cassandra 的外层容器 keyspace，keyspace 中包含了若干列族。keyspace 在逻辑上是容纳列族和某些配置属性的命名空间。列族定义了相关的数据的名字和它们的排序方式。除此之外，数据表都是稀疏的，你可以直接使用需要的列来添加数据，不需要预先定义列的形式。在 Cassandra 里，你不需要使用昂贵的数据建模工具，也不需要写出复杂的包含 join 的查询语句，只需要按照查询的需要来进行建模，之后提供数据给应用即可。

1.3.9 高性能

Cassandra 在设计之初就特别考虑了要充分利用多处理器和多核计算机的性能，并考虑在分布于多个数据中心的大量这类服务器上运行。它可以一致而且无缝地扩展到数百台机器，存储数 TB 的数据。Cassandra 已经显示出了高负载下的良好表现，在一个非常普通的工作站上，Cassandra 也可以提供非常高的写吞吐量。而如果你增加更多的服务器，你还可以继续保持 Cassandra 所有的特性而无需牺牲性能。

1.4 Cassandra来自何方

Cassandra 数据存储系统是一个 Apache 开源项目，位于 http://cassandra.apache.org。2007 年 Facebook 为了解决消息收件箱的搜索问题而开始了 Cassandra 项目，因为当时他们遇到了传统的方法难以解决的超大数据量存储的可扩展性问题。具体说来，项目团队需要处理大量的消息副本、消息的反向索引等不同形式的数据，需要处理很多随机读和并发随机写操作。

这个团队由 Jeff Hammerbacher 领导，核心工程师包括 Avinash Lakshman，Karthik Ranganathan 和搜索团队的工程师 Prashant Malik。源代码在 2008 年 7 月公布到 Google Code 上，成为了一个开源项目。但 2008 年作为 Google Code 的项目时，还只有 Facebook 的工程师在更新项目，未形成社区力量。之后，在 2009 年 3 月，Cassandra 成为了一个 Apache 孵化器项目，并在 2010 年 2 月 17 日成为了 Apache 顶级项目。



Facebook 的 Lakshman 和 Malik 写的关于 Cassandra 的论文“A Decentralized Structured Storage System”可以在这里访问 http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf。

如今的 Cassandra 似乎有某种矛盾性：感觉上 Cassandra 很新很激进，但它植根于很多前人建立和信奉的标准、传统的计算机科学概念和信条中。Cassandra 是一个实用主义的数据库，它并不是特地做得和关系型数据库不同来标新立异，也不是某几个天才的玩物。它是被设计用来解决现有工具还不能解决的实际问题的。它了解之前方法的局限性，并专注于新的面向大规模数据的新世界。

Cassandra 如何得名的

总有人问我 Cassandra 从何得名的，让我有点奇怪。当我听到这个项目的时候，名字并不是我首先想到的问题。不过这确实挺有趣的，特别是对于这个数据库，它的名字确实是有讲究的。

在希腊神话里，Cassandra 是特洛伊国王 Priam 和 Hecuba 王后的女儿。Cassandra 非常美丽，以至于阿波罗给了她预见未来的能力。但当她拒绝阿波罗的爱慕的时候，遭到他的诅咒。从此，她依然可以精确地预知未来，但是不会有任何人相信她。Cassandra 预知了她的特洛伊城终将覆灭，但却无力阻止这一悲剧。Cassandra 分布式数据库就据此命名。我怀疑这个名字有点调侃德尔斐神谕（Delphi Oracle），Oracle 也是用先知命名的数据库。

1.5 Cassandra的应用场景

我们已经介绍了 Cassandra 的主要特点，对 Cassandra 的长处有了一定理解。尽管 Cassandra 设计精巧、功能出色，但也不能胜任所有工作。所以，这里我们来介绍一下 Cassandra 最擅长的领域。

1.5.1 大规模部署

你可能不会开着一辆轻型小卡车去取干洗的衣服，小卡车显然不适合这种工作。

Cassandra 的很多精巧设计都专注于高可用、可调一致性、P2P 协议、无缝扩展等，这些都是 Cassandra 的卖点。这些特性在单节点工作时都是没有意义的，更无法实现它的全部能力。

但是，单节点关系数据库在很多情况下可能正是我们需要的。所以，你需要做一些评估。考虑你期望的流量、吞吐需求以及 SLA 等。关于评估没有什么硬性的指标和要求，但如果你认为有几种关系型数据库可以很好地应付你的流量，提供不错的性能，那可能选关系型数据库更好，简单地说，这是因为 RDBMS 更易于在单机上运行，你也更熟悉。

但是，如果你认为需要至少几个节点才能支撑你的业务，那 Cassandra 就是个不错的选择。如果你的应用可能需要数十个节点，那 Cassandra 可能就是个很棒的选择了。

1.5.2 写密集、统计和分析型工作

考虑一下你的应用的读写比例，Cassandra 是为优异的写吞吐量而特别优化的。

许多早期使用 Cassandra 的产品都用于存储用户状态更新、社交网络、建议 / 评价以及应用统计等。这些都是 Cassandra 很好的应用场景，因为这些应用大都是写多于读的，并且更新可能随时发生并伴有突发的峰值。事实上，支撑应用负载需要很高的多客户线程并发写性能，这正是 Cassandra 的主要特性。

根据项目的 wiki，Cassandra 已经被用于开发了多种不同的应用，包括一个窗口化的时间序列数据库，一个用于文档搜索的反向索引，以及一个分布式任务优先级队列。

1.5.3 地区分布

Cassandra 直接支持多地分布的数据存储。Cassandra 可以很容易配置成将数据分布到多个数据中心的存储方式。如果你有一个全球部署的应用，那么让数据贴近用户会获得不错的性能收益，Cassandra 正适合这种应用场合。

1.5.4 变化的应用

如果你正在“初创阶段”，业务会不断改进，Cassandra 这种没有 Schema 的数据模型可能更适合你。这让你的数据库能更好地跟上业务改进的步伐。

1.6 谁在使用Cassandra

不管怎么说，Cassandra 还是处在初级阶段，在本书写作的时候也还没有达到 1.0 发

布的水平。没有什么易用的图形化的管理工具可用，社区之中，也还有一些内部和外部的有争议的设计问题。但是即使是在开发的早期，它作为一种有发展前景、可用而且稳定的数据存储系统，已经被很多知名的大公司用到了产品之中。



实际上，有一种被称为“从众谬误”的逻辑谬误，即那些很流行、很大众化的东西就被认为是对的。Cassandra 毫无疑问有一个火箭式的上升期，特别是在过去一年里。不过，我仍然认为这些在不同公司的不同产品中的应用至少可以证明 Cassandra 是有用的，而且已经随时可用。

正在使用 Cassandra 的公司还在增长中，这些公司如下。

- Twitter 正在使用 Cassandra 做数据分析。在一篇广为引用的博客里（http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html），Twitter 的主要 Cassandra 工程师 Ryan King 解释道，Twitter 决定不会像开始计划的那样使用 Cassandra 作为其主要的 tweets 存储系统，但还会将 Cassandra 用于多个不同的地方，包括：实时分析，地理和位置信息，以及全部用户信息的数据挖掘。
- Mahalo 使用 Cassandra 作为其主要的近实时数据存储。
- Facebook 还在使用 Cassandra 作为其收件箱的索引，不过使用的是一个非开源分支。[1]
- Digg 也使用 Cassandra 作为主要的近实时数据存储。
- Rackspace 使用它进行云服务、监控和日志存储。
- Reddit 使用它作为持久化的缓存。
- Cloudkick 使用 Cassandra 用于监控统计和分析。
- Ooyala 使用 Cassandra 存储和服务近实时的视频分析数据。
- SimpleGeo 使用 Cassandra 作为实时位置信息的主要存储系统。
- Onespot 使用它作为部分主要数据的存储系统。

Cisco 和 Platform64 也在使用 Cassandra，而且 Comcast 和 bee.tv 也准备使用 Cassandra 来为移动设备提供网上的个性化电视服务。其实还有更多。最后要说的是，这些应用都是真实存在的，各种不同领域的公司都找到了 Cassandra 的应用场合并获得成功。在本书写作的时候，最大的 Cassandra 应用来自于 Facebook，他们在 100 多台机器上存储了超过 50 TB 的数据。

很多公司都在不同的产品项目里评估 Cassandra，而由 Apache Cassandra 项目的主席 Jonathan Ellis 等人成立的公司 Riptano 也于 2010 年 4 月成立了。随着更多的特性、

译注1：根据Facebook的工程师团队在2010年11月对他们发布的新消息系统的介绍，新消息系统是基于HBase而非Cassandra的。

更好的工具以及技术支持方案的加入，可以预期会有更多的人加入使用 Cassandra 的行列。

1.7 小结

本章我们介绍了 Cassandra 的特征、历史和主要特性。我们了解到了哪些公司使用了 Cassandra 和它们的使用目的。我们还回顾了数据库技术的发展史，以此可以从历史角度看待 Cassandra 的价值。

第 2 章

安装 Cassandra

对于立即想要尝鲜的读者，我们将从安装 Cassandra 开始。因为 Cassandra 带来了很多新词汇，安装过程中可能会有些我们不太熟悉的词。不过不用担心，本章的目的是快速搭建一个简单可用的系统。先搭起一个环境。之后我们再在这个基础上从更大的背景下理解 Cassandra。

2.1　安装二进制包

Cassandra 可以从其官方网站 http://cassandra.apache.org 上下载。只要单击首页上的下载最新版本的链接就可以下到 gzip 压缩的 tar 包。编译好的二进制包命名为 apache-cassandra-x.x.x-bin.tar.gz，其中 x.x.x 是版本号。压缩包大约 10 MB 大。

2.1.1　解压缩

最简单的开始方法就是下载已经编译好的二进制包。你可以使用任何常规的 ZIP 工具来解开压缩包。在 Linux 下，gzip 应该是所有发布版预装的工具，而在 Windows 下，你可能需要一个 WinZip 之类的工具。WinZip 是商业软件，如果需要免费软件的话，可以使用 7-Zip，这个工具可以从 http://www.7-zip.org 下载。

使用解压工具。你可能需要分两步来打开压缩文件和 tar 包，一旦得到了一个名为 apache-cassandra-x.x.x 的目录，你就可以运行 Cassandra 了。

2.1.2　里面有什么

解开 tar 包之后，你就可以看到 Cassandra 的二进制发布里面的各个目录了。现在让我们先来花点时间看看里面都有些什么。

- bin

  bin 目录包含了用于运行 Cassandra 的可执行文件以及命令行（CLI）客户端。这个目录中还包含运行 `nodetool` 的脚本，用于监控集群是否被合理配置，并进行各种管理操作。在后面我们会深入介绍 `nodetool`。这个目录还包含 Cassandra 的数据文件 SSTable 与 JSON 相互转换的脚本。

- conf

  这个目录在源码包里也位于这个位置，包含了配置 Cassandra 实例所需的配置文件，这些配置文件有三个主要功能：通过 storage-conf.xml 文件，你可以配置 keyspace 和列族，以此创建存储系统；还有一些文件用于鉴权相关设置；最后 log4j.properties 文件是用来配置日志级别等设置的。在第 6 章里，我们在介绍如何配置 Cassandra 时会看到如何使用它们。

- interface

  对于 0.6 和之前版本的 Cassandra，这个目录里只有一个文件——cassandra.thrift。这个文件用于描述 Cassandra 支持的远程调用（RPC）客户端 API。接口使用 Thrift 格式定义，并提供了一个简单的生成客户端的方法。要快速查看 Cassandra 所支持的所有操作，只要使用一个普通文本编辑器打开这个文件就行了。你可以看到 Cassandra 通过这个接口支持 Java、C++、PHP、Ruby、Python、Perl 以及 C# 等各种客户端。

- javadoc

  这个目录包含了 Java 的 JavaDoc 工具自动生成的文档站点。注意，JavaDoc 仅仅是从 Java 源码里的注释直接生成的，并不是一个非常完善的文档。如果你只希望了解代码的结构，这可能还算是个不错的途径。而且，虽然 Cassandra 是个非常优秀的项目，但代码之中的注释却并不多，所以，你可能会发现 JavaDoc 的帮助非常有限。如果你对 Java 比较熟悉，直接阅读 class 文件可能更有效一些。如果还是要阅读 JavaDoc，那么就用浏览器打开 javadoc/index.html 文件即可。

- lib

  这个目录包含 Cassandra 运行所需的外部库。比如，这里面包含了两个不同的 JSON 串行化库，Google collections 项目，以及一些 Apache 的公共库。这个目录还包含 Thrift 和 Avro RPC 库，用于与 Cassandra 的交互。

2.2 从源码编译

Cassandra 使用 Apache Ant 作为编译脚本语言，并使用 Ivy 插件来进行依赖关系管理。



你可以从 http://ant.apache.org 下载 Ant，只是要编译 Cassandra 的话，不需要单独下载 Ivy。

Ivy 依赖于 Ant，并且编译源码还需要 1.6.0_20 以上版本的 JDK，而不仅是 JRE。如果你看到 Ant 缺少 tools.jar 的话，要么是缺少完整的 JDK，要么是环境变量指向了错误的路径。



如果你想下载最新的系统，那么可以从 Hudson 中获取代码，Hudson 是 Cassandra 使用的持续集成工具。最新的源代码和集成测试信息位于 http://hudson.zones.apache.org/hudson/job/Cassandra/。

如果你是 Git 的忠实用户，还可以使用以下命令来下载 Cassandra 源码的主干分支（只读）：

```
>git clone git://git.apache.org/cassandra.git
```



Git 是 Linus Torvalds 开发的用于管理 Linux 内核开发的源代码管理系统。这个工具目前非常流行，被 Android、Fedora、Ruby on Rails、Perl 等项目以及很多 Cassandra 客户端（第 8 章会进一步介绍）项目使用。如果你使用了诸如 Ubuntu 这样的 Linux 发布版，非常容易获得 Git。只要在终端输入 >apt-get install git 就可以装好并开始使用了。如果要进一步了解，可以访问 http://git-scm.com/。

因为 Ivy 可以处理好所有依赖关系，一旦你获取了源代码，编译 Cassandra 就是一件非常容易的事。只要确定你在源代码的根目录，并执行 `ant` 命令就行，`ant` 会根据当前路径中的 build.xml 文件的内容执行默认的编译目标。后面的事情就全由 Ant 和 Ivy 负责了。要运行 Ant 开始编译源码，只要输入如下命令即可：

```
>ant
```

就这么简单，Ivy 会获取所有必要的依赖库，Ant 会编译大约 350 个源文件，并执行测试。如果一切正常，你可以看到一条 `BUILD SUCCESSFUL` 消息。如果出现意外的话，首先检查你的各种路径设置是否正确，是否有所有需要的工具的最新版本，以及是否下载了一个稳定版本的 Cassandra。你可以通过 Hudson 的集成测试报告来查看下载的源代码是否可以正常编译。



如果你希望看到编译过程中的详细信息，可以在运行 Ant 的时候加上 `-v` 选项，这样它会输出每一个操作的详细信息。

2.2.1 其他编译目标

要编译服务器，只要运行上面的命令就可以了。而这里其实还有一些其他的编译目的，你或许也感兴趣。

- test
  对用户来说，这可能是个最有用的目的，它会自动执行所有单元测试。你还可以从单元测试里面发现不少关于如何和 Cassandra 交互的有用的例子。

- gen-thrift-java
  这个目的用于生成使用 Java 和 Cassandra 交互的 Apache Thrift 客户端接口。

- gen-thrift-py
  这个目的用于生成 Python 用户使用的 Thrift 客户端接口。

- build-jar
  用于生成用来发布的 Java 存档（JAR）文件，可以直接执行 >ant jar。这样就可以完成编译过程，并在 build 目录生成一个名为 apache-cassandra-x.x.x.jar 的文件。

2.2.2 使用Maven编译

Cassandra 的最早的作者们似乎并不怎么关注 Maven，所以早期的 Cassandra 版本里也没有包含 Maven POM 文件。不过，由于越来越多的 Java 开发者开始从 Ant 转向 Maven，而且 Maven 的 IDE 工具支持也日渐强大，如果你非常想用 Maven 的话，可以用一个外部贡献的 pom.xml 文件。

要使用 Maven 编译源代码，需要在 <cassandra-home>/contrib/maven 目录执行这条命令：

```
$ mvn clean install
```

如果你使用 Maven 编译遇到困难的话，可能需要手工下载一些需要的 JAR 包。对于 0.6.3 版本来说，因为一些依赖关系，Maven POM 文件是无法直接使用的，比如 libthrift.jar 在 Maven 仓库中就不可用。



因为 Cassandra 的开发者很少使用 Maven，所以 Maven 也缺少强有力的支持。也就是说，你得小心使用 Maven，因为 Maven POM 经常是不可用的。

2.3 运行Cassandra

在 Cassandra 的早期版本中，运行 Cassandra 服务器之前需要调整使用 Ivy 来设置环

境变量。但现在开发者们已经做了一些非常出色的工作，使得我们可以很容易地直接启动 Cassandra。



Cassandra 需要 J2SE JDK6，最好是 1.6.0_20 或是更新的版本。Cassandra 已经在 Open JDK 和 Sun JDK 上进行了测试。你可以使用 `>java-version` 这条命令来检查 Java 的版本。如果需要 JDK 的话，可以从 http://java.sun.com/javase/downloads 下载。

2.3.1 在Windows平台上运行Cassandra

只要下载了二进制包或是下载了源码包编译之后，就可以运行 Cassandra 服务器了。

首先，需要设置 `JAVA_HOME` 环境变量。要在 Windows 7 上设置环境变量，需要单击“开始”按钮，然后在“我的电脑”上点右键。选择“高级系统设置”，然后单击“环境变量……”按钮。单击“新建……”按钮，创建一个新的系统变量。在变量的名称域输入 `JAVA_HOME`。在变量的值域输入 JDK 的安装位置。可能是类似 C:\Program Files\Java\jdk1.6.0_20 的东西。这里，你是在创建一个新的环境变量，所以需要重新打开一个终端窗口才能让新设置的环境变量生效。要确定环境变量是否设置妥当了，在新的终端窗口输入 `>echo %JAVA_HOME%`。输出的值就是你设置的环境变量。

第一次启动服务器时，Cassandra 会在系统中添加两个目录。第一个是 C:\var\lib\cassandra，这个目录用于存放一些称 commitlog 的数据文件。另一个是 C:\var\log\cassandra，在这个目录里，日志会写到名为 system.log 的文件中去。如果你遇到什么问题，可以查看这些目录中的内容，来看看到底发生了什么。如果你在尝试多个不同版本的 Cassandra，并且不担心丢掉数据的话，可以直接删除这些目录再像上次启动系统一样重新启动系统。

2.3.2 在Linux下运行Cassandra

在 Linux 下运行 Cassandra 和在 Windows 下区别不大。首先确定 `JAVA_HOME` 环境变量设置到 1.6.0_20 或是更新版本的 JDK 上。然后用 gunzip 打开 gzip 压缩的 Cassandra 的 tar 包。最后，创建两个目录用于存放 Cassandra 的数据和日志，并设置恰当的权限，如下所示：

```
ehewitt@morpheus$ cd /home/eben/books/cassandra/dist/apache-cassandra-
    0.7.0-beta1
ehewitt@morpheus$ sudo mkdir -p /var/log/cassandra
ehewitt@morpheus$ sudo chown -R ehewitt /var/log/cassandra
ehewitt@morpheus$ sudo mkdir -p /var/lib/cassandra
```

```
ehewitt@morpheus$ sudo chown -R ehewitt /var/lib/cassandra
```

当然，对于你来说不是 ehewitt，应该替换成你自己的用户名。

2.3.3 启动服务器

不论在任何操作系统中，启动 Cassandra 服务器都要在终端窗口里，进入到解压压缩文件得到的 <cassandra-directory>/bin 目录中去，运行下面的命令来启动服务器。对于第一次安装的系统，你应该看到差不多如下的日志内容：

```
eben@morpheus$ bin/cassandra -f
  INFO 13:23:22,367 DiskAccessMode 'auto' determined to be standard,
       indexAccessMode is standard
  INFO 13:23:22,475 Couldn't detect any schema definitions in local storage.
  INFO 13:23:22,476 Found table data in data directories.
Consider using JMX to call org.apache.cassandra.service.StorageService
    .loadSchemaFromYaml().
  INFO 13:23:22,497 Cassandra version: 0.7.0-beta1
  INFO 13:23:22,497 Thrift API version: 10.0.0
  INFO 13:23:22,498 Saved Token not found. Using qFABQw5XJMvs47lg
  INFO 13:23:22,498 Saved ClusterName not found. Using Test Cluster
  INFO 13:23:22,502 Creating new commitlog segment /var/lib/cassandra/
commitlog/ CommitLog-1282508602502.log
  INFO 13:23:22,507 switching in a fresh Memtable for LocationInfo
       at CommitLogContext( file='/var/lib/cassandra/commitlog/CommitLog-
       1282508602502.log', position=276)
  INFO 13:23:22,510 Enqueuing flush of Memtable-LocationInfo@29857804(178
       bytes, 4 operations)
  INFO 13:23:22,511 Writing Memtable-LocationInfo@29857804(178 bytes, 4
       operations)
  INFO 13:23:22,691 Completed flushing /var/lib/cassandra/data/system/
       LocationInfo-e-1-Data.db
  INFO 13:23:22,701 Starting up server gossip
  INFO 13:23:22,750 Binding thrift service to localhost/127.0.0.1:9160
  INFO 13:23:22,752 Using TFramedTransport with a max frame size of
       15728640 bytes.
  INFO 13:23:22,753 Listening for thrift clients...
  INFO 13:23:22,792 mx4j successfuly loaded
  HttpAdaptor version 3.0.2 started on port 8081
```



-f 参数告诉 Cassandra 停留在前台，而不是作为一个后台进程运行，这样服务器的日志就会输出到标准输出来，我们也就可以在终端窗口看到这些信息了，这对于测试来说非常有用。

祝贺你！你的 Cassandra 服务器应该已经运行起来了，并且得到了一个监听 9160 端口的，名为 Test Cluster 的单节点 Cassandra 集群了。



Cassandra 开发者们在努力工作，以保障 Cassandra 每一个小版本升级和每一个主版本升级后都可以继续读取原有的数据。但是，在版本升级时（即使是小版本的升级），你仍需要确保已经完全提交并清空所有 commit log。

如果你有较早的 Cassandra 版本，那么可能需要现在就清空这些数据目录，这只要启动并运行那个已有的系统就可以了。如果你已经删除了旧的 Cassandra，并希望重新从零开始，那么可以删除 /var/lib/cassandra 和 /var/log/cassandra 两个目录。

2.4 使用命令行界面的客户端

现在，你已经拥有了一个运行着的 Cassandra 集群了，我们来操作一下，确认一切正常吧。在 Linux 下，只要用命令行就可以了。而在 Windows 下，你可能还要多做一点工作。

在 Windows 下，访问 Cassandra 的安装目录，并在这里打开一个终端，运行客户端程序：

```
>bin\cassandra-cli
```

对于 Windows，启动客户端时，你可能会看到这样的出错信息：

```
Starting Cassandra Client
Exception in thread "main" java.lang.NoClassDefFoundError:
org/apache/cassandra/cli/CliMain
```

这可能因为是在 bin 目录里直接启动的 Cassandra，这样，它的 Java classpath 设置并不正确，无法找到 CliMain 文件来启动客户端。你需要首先定义一个 `CASSANDRA_HOME` 环境变量，指向 Cassandra 所在的顶级目录，也就是放置或是编译 Cassandra 的目录，这样就不必关注到底是从哪里启动的 Cassandra 了。



对于如何在 Windows 里设置环境变量，可以参考 2.3.1 节。

要在 Linux 下运行命令行客户端，只要到 Cassandra 的安装目录，并运行 bin 目录里的 `cassandra-cli` 命令：

```
>bin/cassandra-cli
```

Cassandra 的客户端就会开始工作：

```
eben@morpheus$ bin/cassandra-cli
Welcome to cassandra CLI.
```

```
Type 'help' or '?' for help. Type 'quit' or 'exit' to quit.
[default@unknown]
```

现在，你就拥有了一个交互式的 shell，可以发出命令了。

不过，还需要注意的一点是，如果你习惯于 Oracle 的 SQL*Plus 或类似的数据库命令行客户端的话，可能会有些失望。Cassandra 的命令行客户端并不是一个全功能的客户端，它确实只是开发用的。对于开始使用 Cassandra 来说，这也是个不错的途径。因为通过这个客户端你不需要写很多代码就可以检测环境是否可用，以及你与 Cassandra 的交互是否正常。

2.5 基本命令行命令

在开始深入研究 Cassandra 之前，我们来笼统看一下客户端的 API，了解你能给服务器送什么样的命令。我们将看到如何使用基本环境命令，以及如何交互来插入或读取数据。

2.5.1 帮助

要在命令行界面下获得帮助信息，只要敲“?”或“help”就能够看到可用命令的列表，如下列出的只是和元数据与配置相关的命令，关于查看与设置值的其他命令我们将在后面介绍。

```
[default@Keyspace1] help
List of all CLI commands:
?                                                       Display this message.
help                                                       Display this help.
help <command>                         Display detailed, command-specific help.
connect <hostname>/<port>                           Connect to thrift service.
use <keyspace> [<username> 'password']                   Switch to a keyspace.
describe keyspace <keyspacename>                            Describe keyspace.
exit                                                                Exit CLI.
quit                                                                Exit CLI.
show cluster name                                        Display cluster name.
show keyspaces                                         Show list of keyspaces.
show api version                                      Show server API version.
create keyspace <keyspace> [with <att1>=<value1> [and <att2>=<value2> ...]]
                   Add a new keyspace with the specified attribute and
                   value(s).
create column family <cf> [with <att1>=<value1> [and <att2>=<value2> ...]]
                   Create a new column family with the specified
                   attribute and value(s).
drop keyspace <keyspace>                                    Delete a keyspace.
drop column family <cf>                                Delete a column family.
rename keyspace <keyspace> <keyspace_new_name>              Rename a keyspace.
rename column family <cf> <new_name>                   Rename a column family.
```

2.5.2 连接服务器

这样启动客户端并不能自动连接到一个 Cassandra 服务器实例。所以，要想连接到某个服务器还需要使用 connect 命令：

```
eben@morpheus:~/books/cassandra/dist/apache-cassandra-0.7.0-beta1$ bin/
    cassandra-cli
Welcome to cassandra CLI.

Type 'help' or '?' for help. Type 'quit' or 'exit' to quit.
[default@unknown] connect localhost/9160
Connected to: "Test Cluster" on localhost/9160
[default@unknown]
```

也可以直接在启动客户端时，通过在参数里指定主机和端口来指定连接到哪个 Cassandra 服务器实例：

```
eben@morpheus:~/books/cassandra/dist/apache-cassandra-0.7.0-beta1$ bin/
    cassandra-cli localhost/9160
Welcome to cassandra CLI.

Type 'help' or '?' for help. Type 'quit' or 'exit' to quit.
[default@unknown]
```



如果在连接服务器的时候遇到类似这样的出错信息：

```
Exception connecting to localhost/9160 - java.net.Connect-
Exception: Connection refused: connect
```

请确定 Cassandra 服务器实例是否在这个主机和端口上，并且尝试一下是否可以 ping 通这个主机。防火墙也可能会阻止连接到服务器上。同时请检查是不是正在使用上面新的 0.7 的语法，这个语法与老版本有所不同。

上面的命令行显示，你连接到了一个称为“Test Cluster”的 Cassandra 服务器集群。这是因为 `localhost` 上的单节点集群默认就是这样设置的。



在一个生产环境里，请确定在配置文件里去掉 Test Cluster 的字样。

2.5.3 描述环境

在连接到 Test Cluster 这个 Cassandra 服务器实例之后，如果你使用的是二进制发布版，那么应该已经设置好了一个空的 keyspace，或者叫 Cassandra 数据库，来供你试用了。

要看当前正在操作的集群的名字，输入：

```
[default@unknown] show cluster name
```

```
Test Cluster
```

要看集群中有哪些 keyspace 可用，使用如下命令：

```
[default@unknown] show keyspaces
system
```

如果你已经创建了自己的 keyspace，它们也会出现在这里。`system` keyspace 是 Cassandra 系统内部使用的，我们不能在里面存放数据。这样看的话，它的作用类似于微软的 SQL Server 里的 master 和 temp 数据库。这个 keyspace 里存放的内容包括 schema 的定义和运行时对 schema 进行的所有修改。利用这个 keyspace，对 schema 的任何修改都可以基于时间戳的先后顺序传遍整个集群。

要查看系统使用的 API 版本，只要输入：

```
[default@Keyspace1] show api version
10.0.0
```

这里有很多其他的命令可以尝试。现在，我们首先在数据库里添加一些数据，再把它们取出来。

2.5.4 创建keyspace和列族

Cassandra keyspace 大致相当于关系数据库里的一个数据库。它会定义一个或多个列族，列族大致可以对应于关系数据库中的表。当我们不指定 keyspace 来启动命令行客户端时，输出大致如下：

```
>bin/cassandra-cli --host localhost --port 9160
Starting Cassandra Client
Connected to: "Test Cluster" on localhost/9160
Welcome to cassandra CLI.

Type 'help' or '?' for help. Type 'quit' or 'exit' to quit.
[default@unknown]
```

shell 提示符显示为 `default@unknown`，因为我们没有作为某一个用户通过鉴权（鉴权的内容会在第 6 章介绍），而且也没指定 keyspace。



这里鉴权的方式和 MySQL 很类似，如果用过 MySQL 应该不会感到陌生。鉴权与授权的功能实际在本书写作时还在开发中。我们建议把 Cassandra 集群放在防火墙内，以确保安全。

我们首先创建自己的 keyspace，这样就能在里面写入数据了：

```
[default@unknown] create keyspace MyKeyspace with replication_factor=1
ab67bad0-ae2c-11df-b642-e700f669bcfc
```

暂时不要管 replication_factor，后面我们会详细探讨这个设置的。在创建了自己的 keyspace 之后，你可以通过如下命令来进入这个 keyspace：

```
[default@unknown] use MyKeyspace
Authenticated to keyspace: MyKeyspace
[default@MyKeyspace]
```

因为 MyKeyspace 并不要求认证，所以我们被授权访问这个 keyspace 了。

现在可以在这个 keyspace 里面创建列族了。要通过命令行创建列族，可以使用如下命令：

```
[default@MyKeyspace] create column family User
991590d3-ae2e-11df-b642-e700f669bcfc
[default@MyKeyspace]
```

这样就在当前 keyspace 创建了一个使用默认列族设置的名为“User”的列族。我们可以使用命令行客户端的 describe keyspace 命令来查看 keyspace 的描述信息和列族的定义，如下：

```
[default@MyKeyspace] describe keyspace MyKeyspace
Keyspace: MyKeyspace

Column Family Name: User
Column Family Type: Standard
Column Sorted By: org.apache.cassandra.db.marshal.BytesType
flush period: null minutes
------
[default@MyKeyspace]
```

我们后面会来关注 Type、Sorted By 以及 flush period 设置。刚开始，现在这些已经足够了。

2.5.5 读写数据

现在，我们已经有了 keyspace 和列族，将向数据库里面写入一些数据再读取出来。虽然现在还不太明白它是如何工作的，不过这问题不大。我们将在后面的章节来逐步熟悉 Cassandra 的数据模型。现在，你已经拥有了一个 keyspace（数据库），其中有一个列族。对于我们的简单目的来说，直接把列族看做是一个多维有序映射就可以，无需提前进行更多设置。列族里包含列，而列是原子级的数据存储单元。

要写入值，可以使用 set 命令：

```
[default@MyKeyspace] set User['ehewitt']['fname']='Eben'
Value inserted.
[default@MyKeyspace] set User['ehewitt']['email']='me@example.com'
```

```
Value inserted.
[default@MyKeyspace]
```

这里为键值 ehewitt 创建了两个列，分别写入了相应的值。列的名字是 fname 和 email。我们可以使用 count 命令来确认已经为这个键值写入两个列了：

```
[default@MyKeyspace] count User['ehewitt']
2 columns
```

现在数据已经在 Cassandra 里了，让我们来把数据读出来，使用 get 命令：

```
[default@MyKeyspace] get User['ehewitt']
=> (column=666e616d65, value=Eben, timestamp=1282510290343000)
=> (column=656d61696c, value=me@example.com, timestamp=1282510313429000)
Returned 2 results.
```

可以使用 del 命令来删除一列。这里将针对 ehewitt 的行键值（row key）删除 email 列：

```
[default@MyKeyspace] del User['ehewitt']['email']
column removed.
```

现在删除整行来清除掉我们留下的痕迹。同样使用 del 命令，不过这次不指定列的名字了：

```
[default@MyKeyspace] del User['ehewitt']
row removed.
```

要确定这行已经被删除了，可以再查询一次：

```
[default@Keyspace1] get User['ehewitt']
Returned 0 results.
```

2.6 小结

现在，你已经安装好一个 Cassandra 集群并运行起来了。我们已经通过命令行客户端插入和取出了一些数据，在真正深入细节之前，该来看看 Cassandra 的全貌了。

第 3 章

Cassandra 的数据模型

在本章中，我们将尝试理解 Cassandra 的设计目标、数据模型以及一些一般的行为特征。

对于关系型数据库的开发者和管理员来说，理解 Cassandra 的数据模型起先可能有不少困难。一些名词是全新的，比如 keyspace，还有一些词可能有不同的含义，比如“列”。如果你尝试直接去对应 Dynamo 或是 BigTable 的文件，可能也容易混淆，因为虽然 Cassandra 是以它们为蓝本的，但却有自己的模型。

所以，本章将从一些共有的概念入手，然后再去熟悉那些新的名词。之后，我们会进行一些实际的建模，以便帮助各位读者了解如何从关系型数据库跨越到 Cassandra 的世界来。

3.1 关系型数据模型

在关系型数据库中，我们拥有数据库本身，这一般是对应于单个应用的最外层的容器。数据库里包含若干张表。表有名字和一个或多个列，每个列也有名字。当向表里添加数据的时候，要指定每一列对应的值。如果对于某一列，没有值相对应，就要置空（`null`）。这组新的数据项给表添加了一行，之后，如果我们知道这行的唯一标识（主键），还可以读出这行内容；否则，就使用 SQL 查询语句来表达约束条件来匹配这行内容。如果希望更新表中的内容，则既可以更新所有行也可以更新部分行，更新的范围可以使用 SQL 语句中的 `where` 子句来选定。

为了学习 Cassandra，最好暂时先把这些来自关系型数据库的知识抛在脑后。

3.2 简介

本节我们将采用自底向上的方法来理解 Cassandra 的数据模型。

可能你想要的最简单的数据存储机制就是数组或列表了，如图 3-1 所示。



图 3-1：值的列表

如果你永久保存了这个列表，就可以在之后进行查询，但可能需要逐个查看值的内容来判断这些值都表达了什么含义，或者需要永远将每一个值放在列表中固定的地方，之后通过描述数据结构的外部文档来记录每个位置都是什么值。这样，你可能就不得不保存一些空的占位内容（如 null），以便保持列表大小一致，即使某些可选的属性的值并不存在（比如传真号码或是公寓号）。总之，数组是一个简单有用的数据结构，但语义不够丰富。

于是，考虑给列表添加第二个维度：值对应的名称。我们给每个元素一个名称，现在有了一个映射（map）结构，如图 3-2 所示。

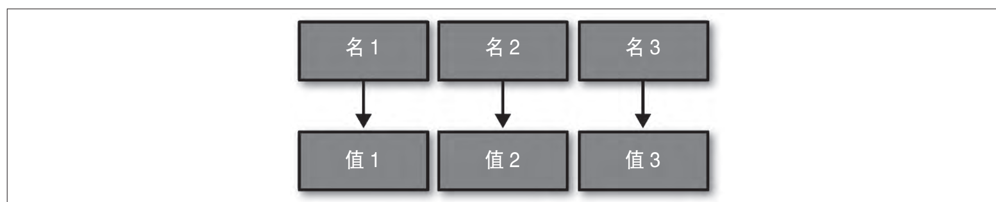


图 3-2："名 / 值"对的映射

现在可以知道一个值的名称了，这当然是一个重大进步。如果我们决定用这个映射来存放用户信息，那么列名应包含 `first name`、`last name`、`phone`、`email` 等，显然，这是一个更丰富的数据结构。

但是，我们建立的数据结构只在描述对象仅拥有一个实例的时候才能工作，比如一个人、用户、宾馆或是一条消息（Tweet）。如果希望在一个数据结构里存放多个对象就不那么容易了，但我们常常就希望这么做。到目前为止，我们还不能创建一个统一的"名 / 值"映射的集合，也无法重复相同的列名。我们需要的实际是把某些列值分成一组，从而可以单独地分组访问。需要一个键值来访问一组列，这一组列可以看做是一个集合。而且还需要行。于是，如果读取一行，就可以获取一个对象

的所有名 / 值对，或者获取我们所关心的名称下的值。我们把每个拥有某组列的集合的对象称为行，每个行的唯一标识称为行键值（row key）。

Cassandra 还定义了一个列族的概念，用作逻辑上的分组，联系起相似的数据。比如，可能有一个用户列族、一个宾馆列族、一个地址簿列族等。依照这个方法，一个列族大致类似于关系数据库领域中的一张表。

把上面提到的这些概念放在一起，就有了 Cassandra 的基本数据结构：列，也就是名 / 值对（客户端还会提供一个最近一次更新的时间戳）；列族，就是为具有相似但不同列集合的行而准备的容器。

关系数据库中，习惯使用字符串来存储列名——这是唯一被允许的。但是在 Cassandra 里，没有类型的限制。行键值和列名都可以和关系型数据库一样是字符串，但也可以是长整数、UUID 或其他任何类型的字节数组。所以，键名的设置就更丰富了。

这揭示了 Cassandra 的列的另一个有趣特质：它们不必是简单的预先设定的“键值对”的关系；你不仅可以在“值”里存放有用数据，在“键”里也同样可以。在 Cassandra 中创建索引时常常就是这样。我们先不必着急往前走这么远。

现在，我们不需要在存储一个新元素时就为每列都存储一个值。也许我们还不知道某个元素每列的值。比如，有些人有第二个电话号码，而很多人没有，而且在一个 Cassandra 支持的在线表单中，可以允许有些域是可选的，有些域是必选的。这些都可以。而且，不知道的值也不必存储一个 `null` 来浪费空间，我们根本不会为某些行存储那些列。这样，就有了如图 3-3 的一个稀疏、多维的数组结构了。

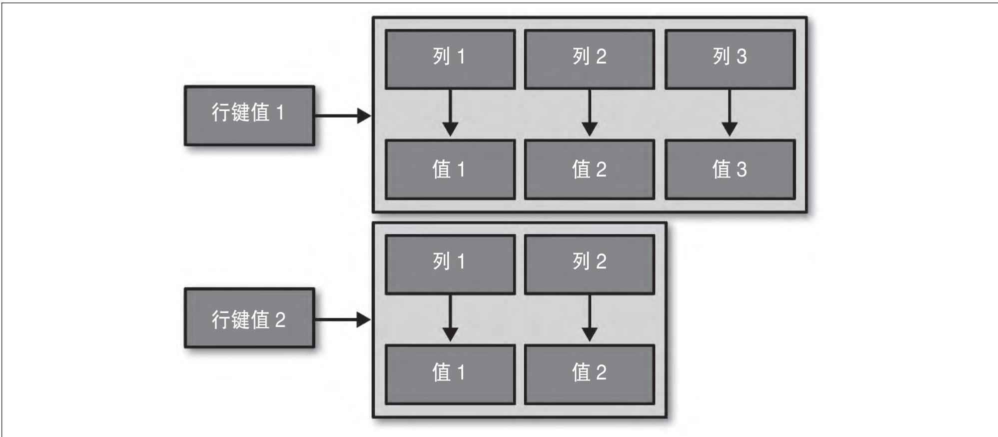


图 3-3：列族

一个 JSON（JavaScript 对象标记）的例子可能比一个图片更有助于理解：

```
Musician:                          ColumnFamily 1
    bootsy:                        RowKey
        email: bootsy@pfunk.com,   ColumnName:Value
        instrument: bass           ColumnName:Value
    george:                        RowKey
        email: george@pfunk.com    ColumnName:Value

Band:                              ColumnFamily 2
    george:                        RowKey
        pfunk: 1968-2010           ColumnName:Value
```

这里有两个列族：音乐家和乐队。音乐家列族有两行——bootsy 和 george。这两行都有一两个列：boosty 有两列（email 和 instrument），而 george 只有一列。这对 Cassandra 来说没什么问题。第二个列族是乐队，里面也有 george 行，并且有一个 pfunk 列。

Cassandra 中的列实际还有第三个元素时间戳，时间戳记录了列上一次被更新的时间。不过，时间戳并不是一个自动的元数据属性，客户端在写数据的时候必须要同时提供时间戳的值。不能通过时间戳查询数据，时间戳仅用于在服务端解决冲突。



行没有时间戳，只有每个单独的列才有时间戳。

那么，如果需要增加一组相关的列该如何做，能不能在这之上增加一个新的维度？Cassandra 允许我们使用*超级列族*（super column family）来完成这个任务。超级列族可以看做是映射的映射。图 3-4 所示的就是超级列族。

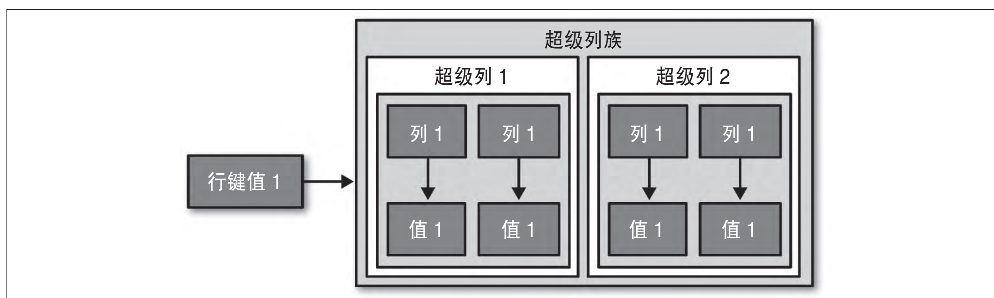


图 3-4：超级列族

一个列族中的一行是一个名 / 值对的集合，而超级列族中的列还包含有一组子列。所以在一个普通的列族里找到一个值可以通过行键值和列名来找到，而在一个类型为“super”的列族中寻址需要行键值、列名和子列的名称。这里其实只有细微的差别，超级列族仍然包含列，只是每个列里拥有子列罢了。

这些就是自底向上看的 Cassandra 的数据模型。现在有了基本的理解，我们可以开足马力上升到更高的层面上，以自顶向下的方法来了解 Cassandra。数据模型这个话

题非常难以理解，所以我们有必要换一个角度来重新描述这个结构，以便帮助读者更全面地理解 Cassandra 的数据模型。

3.3 集群

如果只运行一个单节点，Cassandra 大概不是最佳选择。正如之前提到的，Cassandra 数据库系统是为跨越多台主机共同工作，对用户呈现为一个整体的分布式系统设计的。所以，Cassandra 的最外层结构就是*集群*（cluster），有时也叫做*环*（ring），因为 Cassandra 将集群中的节点组织成一个环，并依此来分配数据到集群中的节点上。

每个节点会存放部分数据的一个副本。如果一个节点宕机了，它的另一个副本可以响应查询请求。P2P 协议允许数据以对用户透明的方式在节点间互相复制，*副本因子*就是所有节点中存放的相同数据的副本数量。我们会在第 6 章里详细讨论这个话题。

3.4 keyspace

集群是 keyspace 的容器，而且里面通常只有一个 keyspace。keyspace 是 Cassandra 中数据的最外层容器，和关系型数据库的概念非常接近。与关系型数据库类似，keyspace 有一个名字和一些定义了整个 keyspace 范围的全局行为的属性。虽然人们常常建议，给每个应用建立一个单独的 keyspace 是个好主意，但实际上这没有太多事实依据。这当然是一个可用的方式，但是按照应用的需要创建足够的 keyspace 才是最好的。可是注意，要是给一个应用创建了上千个 keyspace，恐怕难免会遇到麻烦。

按照选择的分区方式，如果安全限制允许，可以将多个 keyspace 放在同一个集群里。比如，如果应用名叫 Twitter，你可能希望集群叫做 `Twitter-Cluster`，keyspace 叫做 `Twitter`。据我所知，迄今为止还没有广为接受的 Cassandra 命名原则可供遵循。

在 Cassandra 中，可以针对 keyspace 设置的基本属性有如下几个。

- 副本因子（Replication factor）
  简单地说，副本因子就是每行数据会复制到多少个节点上。如果副本因子是 3，那么每行数据将会复制到环上的三个节点，复制过程对于客户端是透明的。
  从本质上说，副本因子的选择决定了要为一致性付出多少性能代价。也就是说，所得到的读写的一致性水平取决于副本因子的设定。

- 副本放置策略（Replica placement strategy）

副本放置策略是指数据的副本如何分布到环上。Cassandra 本身有多种可选的放置策略，用于决定键值到节点的映射方式。这些策略包括：SimpleStrategy（简单策略，之前称为 RackUnawareStrategy，非机架感知策略）、OldNetworkTopologyStrategy（旧网络拓扑策略，之前称为 RackAwareStrategy，机架感知策略）和 NetworkTopology-Strategy（网络拓扑策略，之前称为 DatacenterShardStrategy，数据中心分片策略）。

- 列族（Column families）
  与数据库是表的容器类似，keyspace 是一个或多个列族的容器。列族就类似于关系型数据库里的表，是集合了很多行的容器。每一行都有一些有序的列。列族的设置就呈现了数据结构，每个 keyspace 都至少有一个列族，而通常会有多个列族。

这里，我提到副本因子和副本放置策略是因为它们都是每个 keyspace 的设置。但是，它们并不直接影响数据模型本身。

虽然一般不建议这么做，但可以为一个应用创建多个 keyspace。通常只有希望为不同的列族设置不同的副本因子和副本放置策略的时候，才会考虑让同一个应用使用多个不同 keyspace。比如，对于一些低优先级的数据，可以将它们单独放在一个设有较低副本因子的 keyspace 当中，这样就可以减少一些 Cassandra 复制这些数据的工夫。但是这样或许增加了太多的复杂度，有可能得不偿失。一个更好的选择大概是开始只建立一个 keyspace，视需求再决定是否有必要调整到那个级别。

3.5 列族

列族（column family）是容纳一组有序的行的容器，每行都包含一组有序的列。在关系型数据库世界里，按照模型来建立数据库的时候，会首先指定数据库的名字（对应于 keyspace）和表的名字（有些类似于列族，但不要真的认为两者是一回事，因为事实并非如此），然后定义每张表中列的名字。

有几个不错的原因可以说明白列族和关系型数据库中的表实际相去甚远。首先，Cassandra 被认为是无 schema 的，因为尽管定义了列族，但没定义列，你可以随意在任意列族中添加任意的列，只要需要。其次，列族有两个属性：名称和比较器（comparator）。比较器是在查询数据时返回的列的排序方式，可以根据 long、byte、UTF8 或其他排序方式进行。

在关系型数据库中，表在磁盘上如何排序通常对用户是透明的，而且很少有人会建议根据 RDBMS 如何在磁盘上存储表来进行数据建模的。这是另一个需要时刻注意的列族与表的区别。因为每个列族在磁盘上都存储为不同的文件，所以把相关的列放在同一个列族中十分重要。

列族与关系型数据库的表的另一个不同在于，关系型数据库仅定义了列，而用户提供了值，这就是行。但在 Cassandra 中，一个列族可以放很多个列，甚至可以定义为超级列族。使用超级列族的好处是允许嵌套定义。

标准的列族类型是 Standard，这是默认情况；而对于超级列族，它的类型被设置为 Super。

向一个 Cassandra 列族中写数据的时候，要指定一个或多个列的值。这些值都通过称为行的唯一标识指定。行有唯一的键值，称为行键值（row key），类似于关系型数据库表中的主键，可以唯一标识一行。所以，说 Cassandra 是面向列的并不确切，如果你把行理解为是列的容器倒是更有利于对这一数据模型的理解。这也是为什么一些人认为 Cassandra 的列族类似于四维哈希：

```
[Keyspace][ColumnFamily][Key][Column]
```

我们可以用一种类似 JSON 的方式来表示 Hotel 列族，如下：

```
Hotel {
  key: AZC_043 { name: Cambria Suites Hayden, phone: 480-444-4444,
     address: 400 N. Hayden Rd., city: Scottsdale, state: AZ, zip:
     85255}
  key: AZS_011 { name: Clarion Scottsdale Peak, phone: 480-333-3333,
     address: 3000 N. Scottsdale Rd, city: Scottsdale, state: AZ, zip:
     85255}
  key: CAS_021 { name: W Hotel, phone: 415-222-2222,
     address: 181 3rd Street, city: San Francisco, state: CA, zip:
     94103}
  key: NYN_042 { name: Waldorf Hotel, phone: 212-555-5555,
     address: 301 Park Ave, city: New York, state: NY, zip: 10019}
 }
```



为了简便起见，这里不考虑列的时间戳属性，不过要记住，每个列还有一个时间戳。

在这个例子中，行键值是 Hotel 列族的唯一主键，其中的列包括名称、电话、地址、城市、州和邮政编码。虽然所有这些行恰巧都定义了这些相同的列，但实际上你可以让一行有四列，而另一行有 400 列，并且两者没有任何交叠，这都没什么问题。



同一行的所有数据必须存放在集群中的同一台机器上，这是 Cassandra 的多副本设计的核心要求。这一限制的原因在于每行都有一个关联的行键值，这个键值决定了放置数据副本的位置。更进一步，每列的大小不能超过 2 GB。在设计数据模型的时候，请时刻注意这些限制。

我们可以在命令行客户端用如下方法查询列族：

```
cassandra> get Hotelier.Hotel['NYN_042']
=> (column=zip, value=10019, timestamp=3894166157031651)
=> (column=state, value=NY, timestamp=3894166157031651)
=> (column=phone, value=212-555-5555, timestamp=3894166157031651)
=> (column=name, value=The Waldorf=Astoria, timestamp=3894166157031651)
=> (column=city, value=New York, timestamp=3894166157031651)
=> (column=address, value=301 Park Ave, timestamp=3894166157031651)
Returned 6 results.
```

结果显示，我们有一家在纽约州纽约市的酒店，但因为结果是面向列的，所以有六条结果，对应于这个列族里这行的六个列。注意，尽管这行有六个列，但其他行可以有不同数量的列。

列族选项

你还可以为每个列族定义如下一些附加的参数。

- `keys_cached`
  每个 SSTable 中缓存的位置数量信息。这里指的不是列的名–值对的数量，而是键值的数量，同时列族中行的位置按照最近最少用（LRU）的方式缓存在内存之中。

- `rows_cached`
  缓存进内存中的整行内容的数量，包括每个行键值对应的所有名值对的列表。

- `comment`
  这就是个标准的注释信息，帮你记住列族定义中的重要信息。

- `read_repair_chance`
  一个介于 0 和 1 之间的值。当执行查询操作而没有指定读时检验要求的一致副本数（quorum），致使两个以上副本返回的一行数据的值出现歧义时，这个值决定了进行读修复操作的概率。当读操作比写操作多很多时，你可能会希望这个概率低一些。

- `preload_row_cache`
  指定是否在服务器启动时预读一部分行缓存。

这里我简化了一下这些定义，实际上这些主要是配置信息和服务器行为，与数据模型关系不大。第 6 章还会更详细地介绍这些内容。

3.6 列

列（column）是 Cassandra 数据模型中的最基本数据结构单元。列是一个由名称、

值和时钟构成的三元组，这个时钟可以看做一个时间戳。再次强调，虽然在关系型数据库中，我们非常熟悉列这个名词，但如果你觉得 Cassandra 和关系型数据库里的列是一样的，就大错特错了。首先，在关系型数据库中，你需要通过预先定义所有列的名字来指定表的结构，而在稍后写的时候，则只要根据预先定义好的数据结构提供值就可以了。

然而，在 Cassandra 之中不需要预先定义列，只要在 keyspace 里定义列族，之后就可以开始写数据了。这是因为 Cassandra 之中所有列的名字都是由客户端提供的。这可以让应用在使用数据时灵活得多，也允许数据模型随着时间推移来循序渐进地改变。



Cassandra 的时钟是在 0.7 版本引入的，但今后前途未卜。在 0.7 版本以前，它叫做时间戳，就是一个 Java 的长整型数。现在修改后支持向量时钟（vector clock），这是一种分布式系统中的冲突解决方法，用于 Amazon Dynamo 当中。这就是为什么列的三元组中最后一个既是时间戳，又是时钟了。向量时钟最终可能会成为 Cassandra 0.7 的一部分，也可能不会，尚未最终确定，本书写作时还是 Beta 版。[1]

名称和值的数据类型是 Java 的字节数组，内容经常是字符串。因为名称和值是二进制类型的，所以它们可以是任意长度的。时钟的接口类型是 `org.apache.cassandra.db.IClock`[2]，但 0.7 版本仍然后向兼容之前的时间戳。列的数据结构如图 3-5 所示。

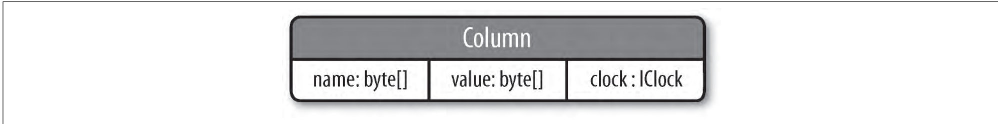


图 3-5：列的数据结构



Cassandra 0.7 引入了一个可选的生存时间（TTL），允许列在创建之后的一段时间后过期。这可能会非常有用。

这是一个列的例子，清晰起见，使用 JSON 格式给出：

```
{
  "name": "email",
```

译注1：最终发布的0.7并不包含向量时钟，该方案已经被最终放弃（Cassandra-580），在下一个版本中，还会包含一个新的冲突解决方案（Cassandra-1072）。

译注2：这一新加入的类型在0.7发布前被撤回了。

```
    "value: "me@example.com",
    "timestamp": 1274654183103300
  }
```

在这个例子中，列的名称是 email，更精确地说，名称属性的值是 email。回忆一下，一个列族可以有多个键值（或行键值），其他的行可能也有这个列。这就是为什么从关系型模型转换到 Cassandra 的模型很困难了：我们总是在想，关系型数据库的每行都有相同的列。但在 Cassandra 里，每个列族有很多行，每个行可以有相同的，也可以有不同的列的集合。

在服务器端，列是不可变的，以防止多线程问题。Cassandra 中，列定义于 `org.apache.cassandra.db.IColumn` 接口，它允许进行很多操作，包括以字节数组类型获取值，或是作为超级列族，以 `Collection<IColumn>` 类型返回子列，以及查找最近改动时间等。

在关系型数据库中，所有的行都存储在一起。早期版本的 Cassandra 并非如此，但 0.6 版本以后的 Cassandra 中，同一个列族的行也都在磁盘上存储在一起。



Cassandra 不支持 join。如果你定义了一个数据模型，发现需要使用 join，那要么不得不在客户端进行，要么创建一个反范式化辅助列族，用于存放 join 的结果。这对于 Cassandra 的用户来说非常普遍。在客户端进行 join 的情况非常少，相反，你真正需要的是复制（反范式化）数据。

3.6.1 宽行与窄行

在为传统的关系型数据库设计表的时候，通常是在处理条目，或者是在处理一个描述特定名词（如宾馆、用户、产品等）的属性集。不会过多考虑行本身的尺寸，因为一旦决定了要在行里放什么，尺寸的问题就没的商量。但是，在使用 Cassandra 时，确实需要决定行的尺寸，它们可宽可窄，依赖于行里的列数。

宽行意味着行有很多很多列（可能包含上万甚至上百万列）。经常会有一些行有很多列。相反情况下，可能会类似关系模型，可以定义很少的列，但会有很多不同的行，这就是窄行模型。

宽行通常容纳自动生成的名字，如 UUID 或时间戳，用于存储一个列表。比如一个监控程序，你可以用一行来存放一个小时的时间片，使用修订的时间戳作为行键值，用列来存储这个时间段内访问应用的 IP 地址。这样，每小时会创建一个新的行键值。

窄行比较类似传统的 RDBMS 行，每行包含了类似的列的名字。与 RDBMS 行的区别在于，所有的列实际都是可选的。

宽行与窄行的另一个区别是，通常只有宽行才会考虑列名的排序问题。下节就讨论这个问题。

3.6.2 列的排序

列的定义之中有另外的一个元素。在 Cassandra 中，在结果返回给客户端的时候，可以指定列的名字如何进行比较和排序。列通过列族中定义的"Compare With"类型来排序，可以从如下类型中选择：`AsciiType`、`BytesType`、`LexicalUUIDType`、`IntegerType`、`LongType`、`TimeUUIDType` 和 `UTF8Type`。

- `AsciiType`
  直接通过比较字节进行排序，每个输入都需要验证是否符合 US-ASCII 编码。US-ASCII 是按照英文字母表的顺序进行编码的。ASCII 定义了 128 个字符，其中 94 个是可打印的。

- `BytesType`
  这是默认的排序方法，直接比较字节，不检查字节的内容是否符合某种编码。以 `BytesType` 作为默认排序方法的一个原因是，对于大部分类型（包括 UTF-8 和 ASCII）来说，按照字节排序都是正确的。

- `LexicalUUIDType`
  一个 16 字节（128 位）的全局唯一标识（UUID），进行字节的排序。

- `LongType`
  按照 8 字节（64 位）的长整型数值进行排序。

- `IntegerType`
  这是 0.7 引入的，比长整型更快，允许使用比 `LongType` 的 64 位更多或更少的位数。

- `TimeUUIDType`
  使用 16 字节（128 位）时间戳进行排序。有五个通用的时间戳 UUID 生成方法。Cassandra 使用的是第一种，这是一种用计算机的 MAC 地址和从公历纪年开始的以 100 纳秒为单位的时间值生成的编号。

- `UTF8Type`
  UTF-8 字符编码的字符串。看起来很适合作为默认排序类型，因为这会让使用 XML 和其他需要公共编码格式的数据交换方式的程序员感到很舒服，但在 Cassandra 中，除非你需要验证数据，否则不要使用 `UTF8Type`。

- `Custom`

  如果愿意，你可以创建自己的排序算法。和 Cassandra 中的很多东西一样，这也是方便易用的，你需要做的就是扩展 `org.apache.cassandra.db.marshal.AbstractType`，并指定自己的类名。

列名的存储是按照 `compare_with` 的值来排序的，行则按照分区器定义的顺序排序存储的（比如使用 RandomPartitioner，就是随机顺序的）。我们会在第 6 章里介绍。

在 Cassandra 之中是无法像关系型数据库那样按照值来排序的。虽然这看起来是个很奇怪的限制，但这是因为 Cassandra 必须按照列名来排序，以便能从一个很宽的行里高效取出一列，而无需把每列都读入内存。性能是 Cassandra 的重要卖点，而实时排序非常影响性能。



列排序是可控的，但键值排序不是，行键值总是按照字节序来排序的。

3.7 超级列

超级列（super column）是一种特殊的列。两种列都是名 / 值对，但普通列的值是字节数组，而超级列的值是一个子列的映射。超级列不能存储其他超级列的映射。也就是说，超级列仅允许使用一层，但是它并不限制列的数量。

超级列的基本数据结构包含它的名字和它存储的列，它的名字和普通的列一样，是字节数组（见图 3-6）。它存储的列保存为一个映射，键值是列的名字，而值是那些列。

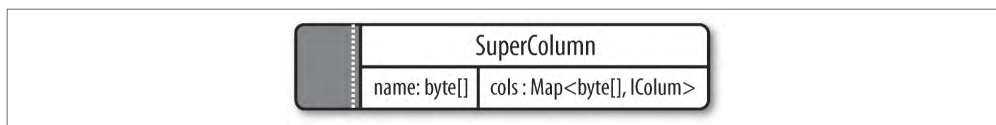


图 3-6：超级列的基本结构

每个列族在磁盘上都存储在自己单独的文件中。所以，要优化 Cassandra 的性能，将经常一起查询的内容保存在同一个列族非常重要，用超级列可以帮助你更容易地做到这点。

`SuperColumn` 类既实现了 `IColumn` 类，也实现了 `IColumnContainer` 类，两者都位于 `org.apache.cassandra.db` 包内。Cassandra 进行远程操作所使用的底层 RPC 序列化机制是 Thrift API。因为 Thrift API 不能标记继承关系，所以，有时 API 会表示

为 ColumnOrSupercolumn 类型，当数据结构使用这个类型表示时，你需要了解外层的列族类型是 Super 还是 Standard。



有趣的事实：超级列是 Facebook 给 Google 的 Bigtable 数据模型增加的更新之一。

这里我们看到了一些更丰富的数据模型。当使用之前看到的普通列时，Cassandra 看起来像是一个四维哈希表。但当引入超级列时，它变成了一个五维哈希：

```
[Keyspace][ColumnFamily][Key][SuperColumn][SubColumn]
```

要使用超级列，就需要定义列族类型为 Super。然后，和用普通列一样，仍然使用行键值，但这个行键值指向的是一个普通列的列表或是映射的名字了，这里的普通列有时也叫做子列。

这里有一个名为 PointOfInterest 的超级列族定义作为例子。在酒店行业中，兴趣点（point of interest）是酒店附近旅客可能乐于造访的地方，比如公园、博物馆、动物园或旅游景点。

```
PointOfInterest (SCF)
    SCkey: Cambria Suites Hayden
    {
         key: Phoenix Zoo
         {
            phone: 480-555-9999,
            desc: They have animals here.
         },
         key: Spring Training
         {
             phone: 623-333-3333,
             desc: Fun for baseball fans.

         },
    }, // Cambria行结束
    SCkey: (UTF8) Waldorf=Astoria
    {
         key: Central Park
            desc: Walk around. It's pretty.
         },
         key: Empire State Building
         {
             phone: 212-777-7777,
             desc: Great view from the 102nd floor.
         }
    }
 }
```

`PointOfInterest`超级列族有两个超级列，每个对应于一个不同的酒店（Cambria Suites Hayden和Waldorf=Astoria），行键值是不同的兴趣点的名字，比如Phoenix Zoo或是Central Park。每行都有一些列作为描述（desc列），有些行有电话号码，而有些没有。这和关系型数据库的表不同，关系型数据库的表里每行结构都是一样的，而列族和超级列里仅仅是一组类似的记录而已。

使用命令行，我们可以查询超级列族：

```
cassandra> get PointOfInterest['Central Park']['The Waldorf=Astoria']
    ['desc'] => (column=desc, value=Walk around in the park.
    It's pretty., timestamp =1281301988847)
```

这个查询是说，在`PointOfInterest`列族（类型是`Super`）中，使用行键值“Central Park”，对于超级列“Waldorf=Astoria”，取“desc”列的值，也就是取这个兴趣点的文字描述。

组合键

使用超级列进行建模的时候，需要考虑一个重要的问题：Cassandra不对子列进行索引，所以，当加载一个超级列进入内存的时候，所有子列都会被载入。



这个问题由Twitter的负责人Ryan King发现，可能会在将来的版本中解决。但这个改动要牵扯到底层存储文件（SSTable），还尚未完成。

现在你可以使用一个自己设置的组合键绕过这个问题。组合键看起来就像`<userid:lastupdate>`。

这些都是在建模时考虑的问题，当你进入到实战阶段的时候会回来检查这些问题。不过，如果数据模型将来可能会有上千个子列，那么你可能就应该采用其他什么方法，而不是超级列了。替代方法之一就是组合键。与使用超级列里的子列不同，组合键使用普通列族里的普通列，只是在键的名字里加入一个自定义的分隔符，在客户端取出的时候分析这个键值就可以了。

这是一个组合键的例子，这个例子还用在了Cassandra具体化视图设计模式里，同时也使用了一种我称之为无值列的Cassandra常用设计模式：

```
HotelByCity (CF) Key: city:state
  key: Phoenix:AZ {AZC_043: -, AZS_011: -}
  key: San Francisco:CA {CAS_021: -}
  key: New York:NY {NYN_042: -}
}
```

这里发生了三件事情。首先，我们在外部定义了一个称为 `Hotel` 的列族，但还可以再创建一个叫做 `HotelByCity` 的列族，存放酒店信息的反范式化数据。我们以不同的方式重复保存了同样的信息，这和 RDBMS 里的视图非常类似，因为它允许我们更快和更直接地进行查询。其次，当要通过城市来查询酒店时（因为很多客户都是通过城市来搜索酒店的），可以创建一个表来定义新的行键值去搜索，但是，不同的州有很多同名的城市（想想 Springfield）[3]，所以我们不能直接使用城市的名字作为行键值，应该加上州的名字。

再后，我们使用另一个称为无值列（valueless column）的模式。我们需要知道的只是城市里有哪些酒店，不需要反范式化更多的东西了。所以，这些列的名字就是它们的值，而不需要对应的值了。这样，在插入列的时候，只需要保存一个空的字节数组作为值就可以了。

3.8 Cassandra与RDBMS的设计差别

Cassandra 的模型和查询方式与 RDBMS 有很多的不同，记住这些差异非常重要。

3.8.1 没有查询语言

SQL 是关系型数据库的标准查询语言，Cassandra 却没有查询语言。不过 Cassandra 确实也有自己的 RPC 序列化机制，Thrift。通过 Thrift API，用户可以访问其中的数据。

3.8.2 没有引用完整性

Cassandra 没有引用完整性的概念，因而没有 join 的概念。在关系型数据库中，你可以在一个表中指定一个外部键值，以此引用另一个表中记录的主键。但是，Cassandra 并没有提供这个功能。存储其他表中的相关 ID 是一个通用需求，这仍然是被支持的，但 Cassandra 里没有级联删除这样的概念。

3.8.3 第二索引

第二索引确实是一个有用的功能，比如你需要找到具有某个属性的酒店的唯一 ID，在关系型数据库里，可能这么查询：

```
SELECT hotelID FROM Hotel WHERE name = 'Clarion Midtown';
```

译注3：美国伊利诺伊州首府，另在俄亥俄州也有同名城市。

当你知道酒店的名字却不知道 ID 的时候，肯定想这么查询这个酒店。关系型数据库如果接到这个查询，会进行一个全表扫描，检查每行的 `name` 列，查找所需要的名字。如果表很大，这种查询可能会很慢。对这种情况，关系型数据库的解决方案就是为这列建一个索引，相当于这部分数据的一个副本，来帮助更快地检索数据。因为 `HotelID` 已经是一个主键约束了，主键会自动进行索引，也就是主索引，所以，对 `name` 列建立的索引自然就是第二索引，目前 Cassandra 仍然不支持第二索引。[4]

要在 Cassandra 中做到同样的事情，需要创建另一个列族来存储查询信息。你可以创建一个列族来存储酒店名，并将它们映射到酒店的 ID。第二列族实际上起到一个显式的第二索引的作用。



第二索引目前正在被加入到 Cassandra 0.7 之中来，允许为列值建立索引。所以，如果你希望找到所有居住在指定城市的用户，第二索引的支持将会让你不必费力手工建立第二索引列族了。

3.8.4 排序成为一种设计决策

在 RDBMS 中，可以在查询中使用 `ORDER BY` 来轻松改变返回记录的顺序。默认的排序方法确实是不可配置的；默认情况下，记录按照它们写入的顺序被读出。如果希望改变顺序，只要改变查询语句即可，而且可以对任意一组列进行排序。但在 Cassandra 之中，排序就不同了，它变成了一个设计决策。列族的定义中包含一个 `CompareWith` 配置元素，这个配置指定了行在读出的时候按照什么方式排序，它在查询的时候是无法重新配置的。

RDBMS 限制你只能基于存储在列中的数据类型来进行排序，但 Cassandra 存储的数据是字节数组，所以这种用指定数据类型排序的方法是行不通的。不过，你能做的是把列当做几种可排序的类型之一（`ASCII`、`LONG`、`integer`、`TimestampUUID`、字典排序等）。如果需要，你还可以使用自己实现的比较器来进行排序。

此外，Cassandra 里没有 SQL 里的 `ORDER BY` 和 `GROUP BY` 语句。有一个查询的类型称为 `SliceRange`，在第 4 章里会介绍到，它类似于 `ORDER BY`，因为它允许翻转。

3.8.5 反范式化

在关系型数据库设计中，我们经常强调范式化的重要性。但是当使用 Cassandra 时，这就不是一个优点了，因为只有当数据模型是反范式化的时候，它的性能才是最好

译注4：如下文提到，Cassandra 0.7加入了对第二索引的支持，该版本在翻译时已经发布了。

的。实际上，很多公司最终都会将关系型数据库反范式化，这主要有两个原因。其一是性能原因，当他们在其多年积累的海量有价值的数据上进行大量的 join 操作的时候，无法得到所需的性能，于是就按照已知的查询内容来反范式化数据库以优化查询。这种方法最终可以工作，但和关系型数据库的设计初衷相悖，最终引发的问题就是，在这种条件下，使用关系型数据库是否还是最佳手段。

关系型数据库进行反范式化的第二个原因是业务文档结构有时需要留存。也就是说，你有一个外围表，引用了很多的外部表，表的数据可能会随时间发生变化，但你也需要以快照形式保存外围文档的历史。常见的一个例子是收款信息。你已经有客户和产品表了，而且认为可以在收款信息里引用这些表。但是实际不应该这么做，因为客户和价格信息都可能发生变化，那时你就会丢失收款信息的完整性了，因为这些表的变动似乎在收款时也发生了，这可能会影响到审计、报告，甚至是违法的，还可能引发其他问题。

在关系型数据库里，反范式化会破坏 Codd 的范式，我们需要尽力避免。但在 Cassandra 中，反范式化却正好合乎规则。它在数据模型很简单时并不必要，但也不需要害怕它。

重点在于，首先对数据建模、然后再写查询的方法不再适用了。Cassandra 中，应该先定义好查询，并围绕查询来组织数据。考虑一下应用使用的最基本的查询路径，之后根据查询路径来构建所需要的列族就可以了。

批评者们认为这是个非常严重的问题。不过在设计数据库的时候能够考虑应用如何查询也并非没有道理，实际上，一般在关系型数据库里也是这么做的。如果不能正确预期查询方式，那么不论是在 Cassandra 里还是在关系型数据库里，都会遇到问题。当然，查询方式可能会随着时间推移而改变，那么就不得不更新数据了。不过这和在关系型数据库里定义表时犯错或需要新的附加表也没什么区别。



有一篇关于 Cloudkick 如何使用 Cassandra 存储性能监控指标数据的文章，可以在这里阅读：https://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra 。

3.9 设计模式

有一些人们通常使用 Cassandra 的方法，可以归纳为设计模式。我给这些模式命名为：具体化视图、无值列和聚合键。

3.9.1 具体化视图

创建一个第二索引来对应附加查询的方法非常常见。因为没有 SQL 的 `WHERE` 子句，就只能通过将数据写到对应于应用查询方式的第二列族的方法来达到这一效果。

比如有一个 `User` 列族，现在你希望通过所在城市来搜索用户，那么就可以创建一个称为 `UserCity` 的第二列族，以城市为键值（而非以用户为键值）存放用户数据，它的列以居住在这个城市的用户的名字来命名。这个反范式化技术将会加快查询，这就是一个围绕查询方式设计数据模型的例子（而非反向进行）。这种使用方式在 Cassandra 中非常常见。当希望通过城市查询用户的时候，只要查询 `UserCity` 列族就可以了，而不用去查询 `User` 列族，然后跨过一个可能很大的数据集，在客户端上做很多琐碎的工作。

注意，在这个语境里，具体化是指在第二列族里存放所有应答查询时需要的原始列族数据的一份完整副本，这样就无需访问原始列族了。如果因为在第二列族里只存放了所需要的名字还要进行第二次查询，相当于第二列族里只存放外部键值，那就是一个第二索引了。



在 0.7 版本里，Cassandra 原生支持了第二索引。

3.9.2 无值列

现在，我们在 `User/UserCity` 例子上继续前进。因为在 `User` 列族中存放了引用信息，这就出现两个问题：首先，需要唯一且完备的键值，以此来达到引用一致性；其次，`UserCity` 列族中的列实际不需要值。如果你有一个行键值 Boise，那么列的名字可以是这个城市里的所有用户名字。因为引用的数据来自于 `User` 列族，所以这些列自己就不需要存储什么值了，你只把它用做一个列表，至于列表中其他相关的附加信息，都可以从被引用的列族中获取。

3.9.3 聚合键

当使用无值列模式时，可能还希望同时使用聚合键模式。这个模式把两个标量值以一个分隔符连在一起来创建一个聚合。现在继续扩展我们的例子。城市的名字常常不具有唯一性，美国的很多州都有叫 Springfield 的城市，得克萨斯州和田纳西州也都有叫巴黎的城市。所以，如果把州的名字和城市的名字融合到一起创建一个聚合键就好多了，可以用在具体化视图之中。这些键值可能会像：`TX:Paris` 和

`TN:Paris` 这样。在传统上，很多 Cassandra 用户都使用冒号作为分隔符，不过使用管道符或是其他的没有歧义的字符都可以。

3.10 需要记住的几件事

在尝试从关系型的思维模式向 Cassandra 的数据模型转变的时候，有几件需要铭记在心的事情。我不得不说：如果你曾经长期使用关系型数据库，那么这个转变并不容易。这里有两点提示。

- 从查询开始。首先要问应用需要什么，围绕应用的需要进行建模，而非像过去在关系型世界里一样，对数据本身进行建模。有些聪明人告诉我，随着业务发展，有新查询的时候，这种方法会让 Cassandra 用户们遇到极大的麻烦。有道理，但对此我要反问一句，为什么要如此死板地定义数据类型，他们的查询并没有这么严格的要求啊。
- 你必须为每个查询提供一个时间戳（或时钟），需要一个策略来同步多个客户端。Cassandra 会使用时间戳来判断哪个是最新写入的值，这非常重要。一个好的策略是使用网络时间协议（NTP）服务器。这里，也有一些聪明人问我，为什么不让服务器自己管理时钟？我的回答是，这是一个对称分布式数据库，服务端实际有同样的问题。

3.11 小结

本章，我们循序渐进地讲解了 Cassandra 的数据模型，包括 keyspace、列族、列以及超级列等概念。我们还了解了一些 RDBMS 与 Cassandra 的不同之处。

第4章

应用实例

本章我们来创建一个完整的实例，这样就可以看到如何把所有东西放在一起。我们将使用各个 API 来看看如何插入数据，进行批量更新，并在列族和超级列族中搜索。

在这个例子中，我们希望使用足够复杂的东西，来展示不同的数据结构和基本的 API 操作，但又不太过复杂让你陷于细枝末节。为了从数据库里得到足够的数据来让搜索正常工作（通过找到所需要的内容、过滤不需要的内容），预装入数据库的内容里会有一些冗余。同时，我希望例子建立在一个大家都熟悉的领域，这样大家能够聚焦在如何使用 Cassandra 工作上，而不是琢磨这个应用是怎么回事。



本章中的代码在 0.7 beta 1 版本中测试通过。当读者使用其他版本时，可能会由于 API 的变动，需要部分的调整。

4.1 数据模型设计

开始构建一个使用关系型数据库的数据驱动新应用时，你可能会首先对应用领域进行建模，构造一系列属性正交的表，并用外部键引用其他表里的相关数据。现在，我们已经明白 Cassandra 是如何存储数据的了，那么就从一个在关系模型里很好理解的小的领域模型开始，然后将它从关系模型映射到 Cassandra 的分布式哈希模型。

简单地说，关系型建模就是从一个概念域开始，然后在表中表达出该域中的名词。

然后分配主键和外部键来对关系进行建模。当遇到多对多的关系时，就需要创建联合表来表达这些键值。联合表在真实世界中并不存在，它们是在关系模型工作时无法避免的副作用。当所有的表都排布好之后，就可以开始写查询了，通过键值定义的关系将分散的数据聚拢到一起。在关系型世界之中，查询是非常次要的东西。关系模型假设，只要正确地建立了表，总能获取到所需要的数据，即使这是以使用很多复杂的子查询或是 join 语句为代价的。

相反，Cassandra 中，不是从数据模型开始的，而要从查询模型开始。

在这个例子里，我们使用一个易于理解、和每个人都有关的领域——允许客人进行预定的酒店。

我们的概念域包括酒店、入住的客人、每个酒店房间的集合以及预定房间的记录，也就是某个客人在某个房间呆的一段时间（称为停留时间）。酒店一般都会保留一个兴趣点的集合，包括公园、博物馆、购物场所、古迹或者位于宾馆附近的某些地方，游客可能会在停留期间前去休闲。宾馆和兴趣点都需要维护地理信息，这样可以在地图上查找，或是计算距离。



显然，在真实世界里要考虑的事情多得多，也复杂得多。比如，酒店的房费总是在不断变化的，计算房费也要考虑各种不同费用情况。这里，我们的定义会足够复杂，让例子足够有趣，也能触及到关键点上，不过也要保持简单，将精力集中在学习 Cassandra 上。

现在来看看如何设计使用 Cassandra 的应用。首先要确定查询。我们希望做如下这些事情：

- 在指定区域查找酒店；
- 查询指定酒店的信息，如名称和位置；
- 查找指定酒店附近的兴趣点；
- 查询指定时间范围的可用房间；
- 查找房间的房费和生活设施；
- 通过输入客户信息来预定选定的房间。

4.2 酒店应用的关系型数据库设计

如图 4-1 所示，我们可以用一个关系型数据库模型来构建这个简单的酒店预定系统。这个关系型模型包含两张联合表，以解决酒店到兴趣点和房间到生活设施之间的多对多关系。

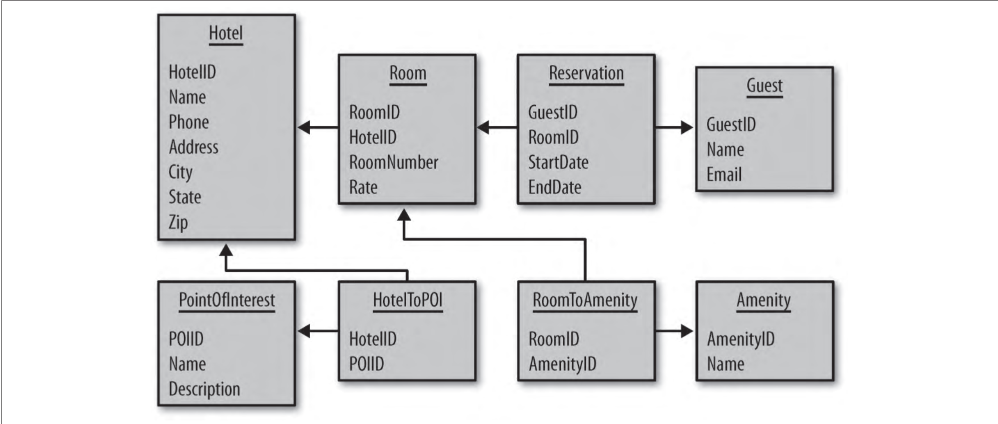


图 4-1：基于 RDBMS 的简单酒店搜索系统

4.3　酒店应用的Cassandra设计

实现这个应用的方法可能不止一种，图 4-2 中，我们给出的是一种使用 Cassandra 物理模型的逻辑数据模型。

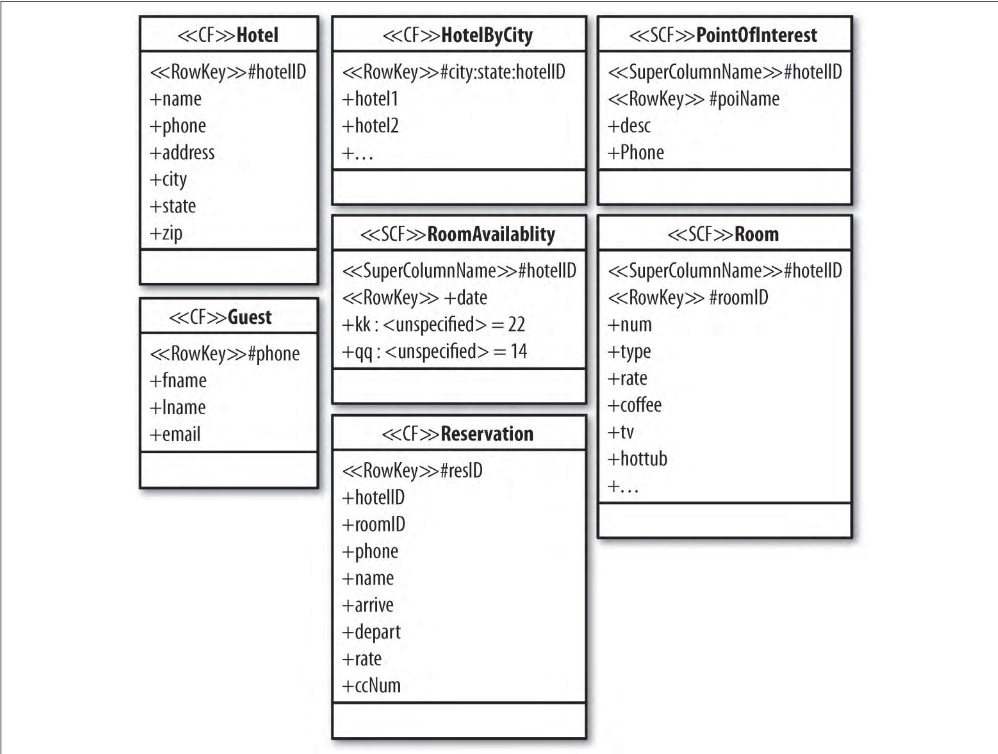


图 4-2：使用 Cassandra 模型的酒店搜索

在 Cassandra 里，我们将完成和之前的关系型模型设计中完全相同的功能。我们直接把 `Hotel`、`Guest` 这几张表转成列族。其他的表，如 `PointOfInterest`，反范式化为一个超级列族。在关系型模型里，你可以用 SQL 通过所在的城市来进行查找。因为 Cassandra 里没有 SQL，我们需要创建 `HotelByCity` 列族来作为索引。

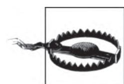


这里，我使用了书名号来标记类型，<<CF>> 表示列族，<<SCF>> 表示超级列族。

我们把房间和设施都放到了 `Room` 列族之中，类型（type）和房费（rate）列分别存储相应的内容，其他如热水浴缸（hot tub）等，如果有相应列名就表示存在相应的设施，否则这列就是空的。

4.4 酒店应用代码

在本节，我们会从头到尾展示一遍代码，来介绍如何实现上述设计。这可以将很多不同的 API 功能的用法展示出来。



这个示例应用的目的就在于展示 Cassandra 之中可以用上的不同思路。这并不是从关系型设计迁移到 Cassandra 模型上的唯一方法。有很多底层的工作是使用 Thrift API 进行的。但 Cassandra（可能）正从 Thrift 向 Avro 进行迁移，所以尽管基本思路没问题，但在实际应用中可能并不能按照这个例子这么做。在第 8 章中，我们介绍了很多可用的第三方 Cassandra 客户端，可以依照你自己的应用开发语言和其他需要，来选择合适的客户端。

创建应用可以分为这么几步。

(1) 构建数据库的结构。
(2) 在数据库中装载酒店和兴趣点的数据。酒店的数据存储在标准的列族中，兴趣点的数据则存储在超级列族中。
(3) 给定城市，搜索一组酒店的列表。这会使用第二索引。
(4) 选择搜索结果中的一个酒店，搜索酒店附近的兴趣点。
(5) 接下来，通过在 `Reservation` 列族插入数据来预定酒店房间的工作应该是水到渠成了，留给读者来完成。

从篇幅上，我们无法实现一个完整的应用，但我们将会完整介绍主要的部分，完成全部实现的工作就是写一些类似的代码而已。

4.4.1 创建数据库

第一步是定义 schema。本例中，我们将使用 YAML 来定义 schema，并载入定义，当然也可以使用客户端代码来进行定义。

YAML 文件定义了主要的 keyspace 和列族，如例 4-1 所示。

例 4-1：cassandra.yaml 中的 schema 定义

```
keyspaces:

    - name: Hotelier
      replica_placement_strategy: org.apache.cassandra.locator.Rack-
          UnawareStrategy replication_factor: 1
      column_families:
        - name: Hotel
          compare_with: UTF8Type

        - name: HotelByCity
          compare_with: UTF8Type

        - name: Guest
          compare_with: BytesType

        - name: Reservation
          compare_with: TimeUUIDType

        - name: PointOfInterest
          column_type: Super
          compare_with: UTF8Type
          compare_subcolumns_with: UTF8Type

        - name: Room
          column_type: Super
          compare_with: BytesType
          compare_subcolumns_with: BytesType

        - name: RoomAvailability
          column_type: Super
          compare_with: BytesType
          compare_subcolumns_with: BytesType
```

这个定义给出了运行这个例子所需的所有列族，而且其中有一些我们在应用代码中并未用到，因为这些是用来完成 RDBMS 设计一样的功能的。

加载schema

在 YAML 里定义好 schema 之后，需要将它们装载到 Cassandra 中。要进行装载，首先打开一个控制台，运行 JDK 提供的 jconsole 工具，使用 JMX 连接到 Cassandra。之后执行 `loadSchemaFromYAML` 操作，这是 `org.apache.cassandra.service.`

StorageService MBean 的一部分。现在 Cassandra 已经知道 schema 了，就可以开始使用 Cassandra 了。你也可以直接使用 API 来创建 keyspace 和列族。

4.4.2 数据结构

我们的应用需要一些标准的数据结构来做为转换对象。这些并不是非常有趣，但确是保持条理性所需要的。我们将使用一个 Hotel 数据结构来保存所有有关酒店的信息，如例 4-2 所示。

例 4-2：Hotel.java

```
package com.cassandraguide.hotel;

//数据传输对象
public class Hotel {
    public String id;
    public String name;
    public String phone;
    public String address;
    public String city;
    public String state;
    public String zip;
}
```

这个结构就是用来保存相应的列信息，以方便应用使用。

我们还有一个 POI 数据结构，用于保存兴趣点的信息，如例 4-3 所示。

例 4-3：POI.java

```
package com.cassandraguide.hotel;

//兴趣点的数据传输用对象
public class POI {
     public String name;
     public String desc;
     public String phone;
}
```

还有一个 Constants 类，把常用的字符串放在一起，易于修改，如例 4-4 所示。

例 4-4：Constants.java

```
package com.cassandraguide.hotel;

import org.apache.cassandra.thrift.ConsistencyLevel;

public class Constants {

    public static final String CAMBRIA_NAME = "Cambria Suites Hayden";
    public static final String CLARION_NAME= "Clarion Scottsdale Peak";
```

```
    public static final String W_NAME = "The W SF";
    public static final String WALDORF_NAME = "The Waldorf=Astoria";

    public static final String UTF8 = "UTF8";
    public static final String KEYSPACE = "Hotelier";
    public static final ConsistencyLevel CL = ConsistencyLevel.ONE;
    public static final String HOST = "localhost";
    public static final int PORT = 9160;
}
```

把常用的字符串保存在一起会让代码更加干净整洁，而且你也可以方便地修改这些字符串，来适应应用环境。

4.4.3 进行连接

为了方便，也为了省去大量看枯燥无味的代码的时间，我们把连接管理的代码放到一个类里面，叫做 Connector，如例 4-5 所示。

例 4-5：一个客户端连接辅助类，Connector.java

```
package com.cassandraguide.hotel;

import static com.cassandraguide.hotel.Constants.KEYSPACE;

import org.apache.cassandra.thrift.Cassandra;
import org.apache.cassandra.thrift.InvalidRequestException;
import org.apache.thrift.TException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;
import org.apache.thrift.transport.TTransportException;

//简单的辅助类，用于封装连接代码，减少重复输入
public class Connector {

    TTransport tr = new TSocket("localhost", 9160);

    // 返回一个到 keyspace 的新连接
    public Cassandra.Client connect() throws TTransportException,
           TException, InvalidRequestException {

        TFramedTransport tf = new TFramedTransport(tr);
        TProtocol proto = new TBinaryProtocol(tf);
        Cassandra.Client client = new Cassandra.Client(proto);
        tr.open();
        client.set_keyspace(KEYSPACE);
        return client;
    }

    public void close() {
```

```
            tr.close();
        }
    }
```

当需要执行数据库操作时，可以使用这个类来打开一个连接，当操作完成时也可以使用这个类来关闭连接。

4.4.4 预装填数据库

例 4-6 所示的 `Prepopulate` 类，进行大批量的 `insert` 和 `batch_mutate` 操作，将供用户查询的酒店和兴趣点的信息预装入到数据库当中。

例 4-6：Prepopulate.java

```
package com.cassandraguide.hotel;

import static com.cassandraguide.hotel.Constants.CAMBRIA_NAME;
import static com.cassandraguide.hotel.Constants.CL;
import static com.cassandraguide.hotel.Constants.CLARION_NAME;
import static com.cassandraguide.hotel.Constants.UTF8;
import static com.cassandraguide.hotel.Constants.WALDORF_NAME;
import static com.cassandraguide.hotel.Constants.W_NAME;

import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Cassandra;
import org.apache.cassandra.thrift.Clock;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ColumnPath;
import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SuperColumn;
import org.apache.log4j.Logger;

/**
 * 装载数据库的初始信息
 * 装填列族和超级列族，包括酒店、兴趣点和索引数据
 * 展示如何使用列族和超级列族的 batch_mutate 和 insert 操作
 *
 * 为节省空间，省略了代码中的所有异常相关操作
 */
public class Prepopulate {
    private static final Logger LOG = Logger.getLogger(Prepopulate.
        class);

    private Cassandra.Client client;
    private Connector connector;
```

```
// 在构造器中打开连接，这样就不必重复打开连接了
public Prepopulate() throws Exception {
    connector = new Connector();
    client = connector.connect();
}

void prepopulate() throws Exception {
    // 预装载酒店数据库
    insertAllHotels();

    // 添加酒店的索引，方便查询
    insertByCityIndexes();

    // 预装载兴趣点数据库
    insertAllPointsOfInterest();

    connector.close();
}

// 添加通过城市查找酒店的索引
public void insertByCityIndexes() throws Exception  {

    String scottsdaleKey = "Scottsdale:AZ";
    String sfKey = "San Francisco:CA";
    String newYorkKey = "New York:NY";

    insertByCityIndex(scottsdaleKey, CAMBRIA_NAME);
    insertByCityIndex(scottsdaleKey, CLARION_NAME);
    insertByCityIndex(sfKey, W_NAME);
    insertByCityIndex(newYorkKey, WALDORF_NAME);
}

// 使用无值列模式
private void insertByCityIndex(String rowKey, String hotelName)
    throws Exception {

    Clock clock = new Clock(System.nanoTime());

    Column nameCol = new Column(hotelName.getBytes(UTF8),
            new byte[0], clock);

    ColumnOrSuperColumn nameCosc = new ColumnOrSuperColumn();
    nameCosc.column = nameCol;

    Mutation nameMut = new Mutation();
    nameMut.column_or_supercolumn = nameCosc;

    // 设置 batch
    Map<String, Map<String, List<Mutation>>> mutationMap =
         new HashMap<String, Map<String, List<Mutation>>>();

    Map<String, List<Mutation>> muts =
         new HashMap<String, List<Mutation>>();
    List<Mutation> cols = new ArrayList<Mutation>();
    cols.add(nameMut);
```

```
    String columnFamily = "HotelByCity";
    muts.put(columnFamily, cols);

    // 外层映射的键值是行键值
    // 内层映射的键值是列族名
    mutationMap.put(rowKey, muts);

   // 创建列的描述
    ColumnPath cp = new ColumnPath(columnFamily);
    cp.setColumn(hotelName.getBytes(UTF8));

    ColumnParent parent = new ColumnParent(columnFamily);
    // 这里，列名就是它的内容本身（所以值是空的）
    Column col = new Column(hotelName.getBytes(UTF8), new byte[0],
        clock);

    client.insert(rowKey.getBytes(), parent, col, CL);

    LOG.debug("Inserted HotelByCity index for " + hotelName);

} // 插入 ByCity 索引结束

//POI：兴趣点
public void insertAllPointsOfInterest() throws Exception {

    LOG.debug("Inserting POIs.");

    insertPOIEmpireState();
    insertPOICentralPark();
    insertPOIPhoenixZoo();
    insertPOISpringTraining();

    LOG.debug("Done inserting POIs.");
}

private void insertPOISpringTraining() throws Exception {
    //Map<byte[],Map<String, List<Mutation>>>
    Map<byte[], Map<String, List<Mutation>>> outerMap =
        new HashMap<byte[], Map<String, List<Mutation>>>();
    List<Mutation> columnsToAdd = new ArrayList<Mutation>();

    Clock clock = new Clock(System.nanoTime());
    String keyName = "Spring Training";
    Column descCol = new Column("desc".getBytes(UTF8),
        "Fun for baseball fans.".getBytes("UTF-8"), clock);
    Column phoneCol = new Column("phone".getBytes(UTF8),
        "623-333-3333".getBytes(UTF8), clock);

    List<Column> cols = new ArrayList<Column>();
    cols.add(descCol);
    cols.add(phoneCol);

    Map<String, List<Mutation>> innerMap =
        new HashMap<String, List<Mutation>>();
```

```
        Mutation columns = new Mutation();
        ColumnOrSuperColumn descCosc = new ColumnOrSuperColumn();
        SuperColumn sc = new SuperColumn();
        sc.name = CAMBRIA_NAME.getBytes();
        sc.columns = cols;

        descCosc.super_column = sc;
        columns.setColumn_or_supercolumn(descCosc);

        columnsToAdd.add(columns);

        String superCFName = "PointOfInterest";
        ColumnPath cp = new ColumnPath();
        cp.column_family = superCFName;
        cp.setSuper_column(CAMBRIA_NAME.getBytes());
        cp.setSuper_columnIsSet(true);

        innerMap.put(superCFName, columnsToAdd);
        outerMap.put(keyName.getBytes(), innerMap);

        client.batch_mutate(outerMap, CL);

        LOG.debug("Done inserting Spring Training.");
    }

    private void insertPOIPhoenixZoo() throws Exception {

        Map<byte[], Map<String, List<Mutation>>> outerMap =
            new HashMap<byte[], Map<String, List<Mutation>>>();
        List<Mutation> columnsToAdd = new ArrayList<Mutation>();

        long ts = System.currentTimeMillis();
        String keyName = "Phoenix Zoo";
        Column descCol = new Column("desc".getBytes(UTF8),
            "They have animals here.".getBytes("UTF-8"), new Clock(ts));

        Column phoneCol = new Column("phone".getBytes(UTF8),
            "480-555-9999".getBytes(UTF8), new Clock(ts));

        List<Column> cols = new ArrayList<Column>();
        cols.add(descCol);
        cols.add(phoneCol);

        Map<String, List<Mutation>> innerMap =
            new HashMap<String, List<Mutation>>();

        String cambriaName = "Cambria Suites Hayden";

        Mutation columns = new Mutation();
        ColumnOrSuperColumn descCosc = new ColumnOrSuperColumn();
        SuperColumn sc = new SuperColumn();
        sc.name = cambriaName.getBytes();
        sc.columns = cols;

        descCosc.super_column = sc;
```

```
    columns.setColumn_or_supercolumn(descCosc);

    columnsToAdd.add(columns);

    String superCFName = "PointOfInterest";
    ColumnPath cp = new ColumnPath();
    cp.column_family = superCFName;
    cp.setSuper_column(cambriaName.getBytes());
    cp.setSuper_columnIsSet(true);

    innerMap.put(superCFName, columnsToAdd);
    outerMap.put(keyName.getBytes(), innerMap);

    client.batch_mutate(outerMap, CL);

    LOG.debug("Done inserting Phoenix Zoo.");
}

private void insertPOICentralPark() throws Exception {

    Map<byte[], Map<String, List<Mutation>>> outerMap =
        new HashMap<byte[], Map<String, List<Mutation>>>();
    List<Mutation> columnsToAdd = new ArrayList<Mutation>();

    Clock clock = new Clock(System.nanoTime());
    String keyName = "Central Park";
    Column descCol = new Column("desc".getBytes(UTF8),
         "Walk around in the park. It's pretty.".getBytes("UTF-8"),
         clock);

    //公园没有电话列
    List<Column> cols = new ArrayList<Column>();
    cols.add(descCol);

    Map<String, List<Mutation>> innerMap =
        new HashMap<String, List<Mutation>>();

    Mutation columns = new Mutation();
    ColumnOrSuperColumn descCosc = new ColumnOrSuperColumn();
    SuperColumn waldorfSC = new SuperColumn();
    waldorfSC.name = WALDORF_NAME.getBytes();
    waldorfSC.columns = cols;

    descCosc.super_column = waldorfSC;
    columns.setColumn_or_supercolumn(descCosc);

    columnsToAdd.add(columns);

    String superCFName = "PointOfInterest";
    ColumnPath cp = new ColumnPath();
    cp.column_family = superCFName;
    cp.setSuper_column(WALDORF_NAME.getBytes());
    cp.setSuper_columnIsSet(true);

    innerMap.put(superCFName, columnsToAdd);
```

```
        outerMap.put(keyName.getBytes(), innerMap);

        client.batch_mutate(outerMap, CL);

        LOG.debug("Done inserting Central Park.");
    }

    private void insertPOIEmpireState()  throws Exception {

        Map<byte[], Map<String, List<Mutation>>> outerMap =
             new HashMap<byte[], Map<String, List<Mutation>>>();

        List<Mutation> columnsToAdd = new ArrayList<Mutation>();

        Clock clock = new Clock(System.nanoTime());
        String esbName = "Empire State Building";
        Column descCol = new Column("desc".getBytes(UTF8),
            "Great view from 102nd floor.".getBytes("UTF-8"),
            clock);
        Column phoneCol = new Column("phone".getBytes(UTF8),
            "212-777-7777".getBytes(UTF8), clock);

        List<Column> esbCols = new ArrayList<Column>();
        esbCols.add(descCol);
        esbCols.add(phoneCol);

        Map<String, List<Mutation>> innerMap = new HashMap<String, List
            <Mutation>>();

        Mutation columns = new Mutation();
        ColumnOrSuperColumn descCosc = new ColumnOrSuperColumn();
        SuperColumn waldorfSC = new SuperColumn();

        waldorfSC.name = WALDORF_NAME.getBytes();
        waldorfSC.columns = esbCols;

        descCosc.super_column = waldorfSC;
        columns.setColumn_or_supercolumn(descCosc);

        columnsToAdd.add(columns);

        String superCFName = "PointOfInterest";
        ColumnPath cp = new ColumnPath();
        cp.column_family = superCFName;
        cp.setSuper_column(WALDORF_NAME.getBytes());
        cp.setSuper_columnIsSet(true);

        innerMap.put(superCFName, columnsToAdd);
        outerMap.put(esbName.getBytes(), innerMap);

        client.batch_mutate(outerMap, CL);

        LOG.debug("Done inserting Empire State.");
    }
```

```
// 用于一次进行所有插入操作的辅助方法
public void insertAllHotels() throws Exception {

    String columnFamily = "Hotel";

    // 行键值
    String cambriaKey = "AZC_043";
    String clarionKey = "AZS_011";
    String wKey = "CAS_021";
    String waldorfKey = "NYN_042";

    // 辅助操作
    Map<byte[], Map<String, List<Mutation>>> cambriaMutationMap =
        createCambriaMutation(columnFamily, cambriaKey);

    Map<byte[], Map<String, List<Mutation>>> clarionMutationMap =
        createClarionMutation(columnFamily, clarionKey);

    Map<byte[], Map<String, List<Mutation>>> waldorfMutationMap =
        createWaldorfMutation(columnFamily, waldorfKey);

    Map<byte[], Map<String, List<Mutation>>> wMutationMap =
        createWMutation(columnFamily, wKey);

    client.batch_mutate(cambriaMutationMap, CL);
    LOG.debug("Inserted " + cambriaKey);
    client.batch_mutate(clarionMutationMap, CL);
    LOG.debug("Inserted " + clarionKey);
    client.batch_mutate(wMutationMap, CL);
    LOG.debug("Inserted " + wKey);
    client.batch_mutate(waldorfMutationMap, CL);
    LOG.debug("Inserted " + waldorfKey);

    LOG.debug("Done inserting at " + System.nanoTime());
}

// 设置要插入的 W 的各列
private Map<byte[], Map<String, List<Mutation>>> createWMutation(
    String columnFamily, String rowKey)
    throws UnsupportedEncodingException {

    Clock clock = new Clock(System.nanoTime());

    Column nameCol = new Column("name".getBytes(UTF8),
        W_NAME.getBytes("UTF-8"), clock);
    Column phoneCol = new Column("phone".getBytes(UTF8),
        "415-222-2222".getBytes(UTF8), clock);
    Column addressCol = new Column("address".getBytes(UTF8),
        "181 3rd Street".getBytes(UTF8), clock);
    Column cityCol = new Column("city".getBytes(UTF8),
        "San Francisco".getBytes(UTF8), clock);
    Column stateCol = new Column("state".getBytes(UTF8),
        "CA".getBytes("UTF-8"), clock);
    Column zipCol = new Column("zip".getBytes(UTF8),
        "94103".getBytes(UTF8), clock);
```

```
        ColumnOrSuperColumn nameCosc = new ColumnOrSuperColumn();
        nameCosc.column = nameCol;

        ColumnOrSuperColumn phoneCosc = new ColumnOrSuperColumn();
        phoneCosc.column = phoneCol;

        ColumnOrSuperColumn addressCosc = new ColumnOrSuperColumn();
        addressCosc.column = addressCol;

        ColumnOrSuperColumn cityCosc = new ColumnOrSuperColumn();
        cityCosc.column = cityCol;

        ColumnOrSuperColumn stateCosc = new ColumnOrSuperColumn();
        stateCosc.column = stateCol;

        ColumnOrSuperColumn zipCosc = new ColumnOrSuperColumn();
        zipCosc.column = zipCol;

        Mutation nameMut = new Mutation();
        nameMut.column_or_supercolumn = nameCosc;
        Mutation phoneMut = new Mutation();
        phoneMut.column_or_supercolumn = phoneCosc;
        Mutation addressMut = new Mutation();
        addressMut.column_or_supercolumn = addressCosc;
        Mutation cityMut = new Mutation();
        cityMut.column_or_supercolumn = cityCosc;
        Mutation stateMut = new Mutation();
        stateMut.column_or_supercolumn = stateCosc;
        Mutation zipMut = new Mutation();
        zipMut.column_or_supercolumn = zipCosc;

        //设置batch
        Map<byte[], Map<String, List<Mutation>>> mutationMap =
             new HashMap<byte[], Map<String, List<Mutation>>>();

        Map<String, List<Mutation>> muts =
             new HashMap<String, List<Mutation>>();
        List<Mutation> cols = new ArrayList<Mutation>();
        cols.add(nameMut);
        cols.add(phoneMut);
        cols.add(addressMut);
        cols.add(cityMut);
        cols.add(stateMut);
        cols.add(zipMut);

        muts.put(columnFamily, cols);

        //外层映射是行键值
        //内层映射是列族名
        mutationMap.put(rowKey.getBytes(), muts);
        return mutationMap;
    }

    //添加Waldorf酒店到Hotel列族
```

```
private Map<byte[], Map<String, List<Mutation>>> createWaldorfMutation(
        String columnFamily, String rowKey)
        throws UnsupportedEncodingException {

    Clock clock = new Clock(System.nanoTime());

    Column nameCol = new Column("name".getBytes(UTF8),
        WALDORF_NAME.getBytes("UTF-8"), clock);
    Column phoneCol = new Column("phone".getBytes(UTF8),
        "212-555-5555".getBytes(UTF8), clock);
    Column addressCol = new Column("address".getBytes(UTF8),
        "301 Park Ave".getBytes(UTF8), clock);
    Column cityCol = new Column("city".getBytes(UTF8),
        "New York".getBytes(UTF8), clock);
    Column stateCol = new Column("state".getBytes(UTF8),
        "NY".getBytes("UTF-8"), clock);
    Column zipCol = new Column("zip".getBytes(UTF8),
        "10019".getBytes(UTF8), clock);

    ColumnOrSuperColumn nameCosc = new ColumnOrSuperColumn();
    nameCosc.column = nameCol;

    ColumnOrSuperColumn phoneCosc = new ColumnOrSuperColumn();
    phoneCosc.column = phoneCol;

    ColumnOrSuperColumn addressCosc = new ColumnOrSuperColumn();
    addressCosc.column = addressCol;

    ColumnOrSuperColumn cityCosc = new ColumnOrSuperColumn();
    cityCosc.column = cityCol;

    ColumnOrSuperColumn stateCosc = new ColumnOrSuperColumn();
    stateCosc.column = stateCol;

    ColumnOrSuperColumn zipCosc = new ColumnOrSuperColumn();
    zipCosc.column = zipCol;

    Mutation nameMut = new Mutation();
    nameMut.column_or_supercolumn = nameCosc;
    Mutation phoneMut = new Mutation();
    phoneMut.column_or_supercolumn = phoneCosc;
    Mutation addressMut = new Mutation();
    addressMut.column_or_supercolumn = addressCosc;
    Mutation cityMut = new Mutation();
    cityMut.column_or_supercolumn = cityCosc;
    Mutation stateMut = new Mutation();
    stateMut.column_or_supercolumn = stateCosc;
    Mutation zipMut = new Mutation();
    zipMut.column_or_supercolumn = zipCosc;

    //设置 batch
    Map<byte[], Map<String, List<Mutation>>> mutationMap =
        new HashMap<byte[], Map<String, List<Mutation>>>();

    Map<String, List<Mutation>> muts =
```

```
        new HashMap<String, List<Mutation>>();
    List<Mutation> cols = new ArrayList<Mutation>();
    cols.add(nameMut);
    cols.add(phoneMut);
    cols.add(addressMut);
    cols.add(cityMut);
    cols.add(stateMut);
    cols.add(zipMut);

    muts.put(columnFamily, cols);

    //外层映射是行键值
    //内层映射是列族名
    mutationMap.put(rowKey.getBytes(), muts);
    return mutationMap;
}

//设置 Clarion 要插入的各列
private Map<byte[], Map<String, List<Mutation>>> createClarionMutation(
        String columnFamily, String rowKey)
        throws UnsupportedEncodingException {

    Clock clock = new Clock(System.nanoTime());

    Column nameCol = new Column("name".getBytes(UTF8),
        CLARION_NAME.getBytes("UTF-8"), clock);
    Column phoneCol = new Column("phone".getBytes(UTF8),
        "480-333-3333".getBytes(UTF8), clock);
    Column addressCol = new Column("address".getBytes(UTF8),
        "3000 N. Scottsdale Rd".getBytes(UTF8), clock);
    Column cityCol = new Column("city".getBytes(UTF8),
        "Scottsdale".getBytes(UTF8), clock);
    Column stateCol = new Column("state".getBytes(UTF8),
        "AZ".getBytes("UTF-8"), clock);
    Column zipCol = new Column("zip".getBytes(UTF8),
        "85255".getBytes(UTF8), clock);

    ColumnOrSuperColumn nameCosc = new ColumnOrSuperColumn();
    nameCosc.column = nameCol;

    ColumnOrSuperColumn phoneCosc = new ColumnOrSuperColumn();
    phoneCosc.column = phoneCol;

    ColumnOrSuperColumn addressCosc = new ColumnOrSuperColumn();
    addressCosc.column = addressCol;

    ColumnOrSuperColumn cityCosc = new ColumnOrSuperColumn();
    cityCosc.column = cityCol;

    ColumnOrSuperColumn stateCosc = new ColumnOrSuperColumn();
    stateCosc.column = stateCol;

    ColumnOrSuperColumn zipCosc = new ColumnOrSuperColumn();
    zipCosc.column = zipCol;
```

```
        Mutation nameMut = new Mutation();
        nameMut.column_or_supercolumn = nameCosc;
        Mutation phoneMut = new Mutation();
        phoneMut.column_or_supercolumn = phoneCosc;
        Mutation addressMut = new Mutation();
        addressMut.column_or_supercolumn = addressCosc;
        Mutation cityMut = new Mutation();
        cityMut.column_or_supercolumn = cityCosc;
        Mutation stateMut = new Mutation();
        stateMut.column_or_supercolumn = stateCosc;
        Mutation zipMut = new Mutation();
        zipMut.column_or_supercolumn = zipCosc;

        //设置 batch
        Map<byte[], Map<String, List<Mutation>>> mutationMap =
            new HashMap<byte[], Map<String, List<Mutation>>>();

        Map<String, List<Mutation>> muts =
            new HashMap<String, List<Mutation>>();
        List<Mutation> cols = new ArrayList<Mutation>();
        cols.add(nameMut);
        cols.add(phoneMut);
        cols.add(addressMut);
        cols.add(cityMut);
        cols.add(stateMut);
        cols.add(zipMut);

        muts.put(columnFamily, cols);

        //外层映射是行键值
        //内层映射是列族名
        mutationMap.put(rowKey.getBytes(), muts);
        return mutationMap;
    }

    //设置要插入的 Cambria 的各列
    private Map<byte[], Map<String, List<Mutation>>> createCambriaMutation(
            String columnFamily, String cambriaKey)
            throws UnsupportedEncodingException {

        //设置 Cambria 的各列
        Clock clock = new Clock(System.nanoTime());

        Column cambriaNameCol = new Column("name".getBytes(UTF8),
            "Cambria Suites Hayden".getBytes("UTF-8"), clock);
        Column cambriaPhoneCol = new Column("phone".getBytes(UTF8),
            "480-444-4444".getBytes(UTF8), clock);
        Column cambriaAddressCol = new Column("address".getBytes(UTF8),
            "400 N. Hayden".getBytes(UTF8), clock);
        Column cambriaCityCol = new Column("city".getBytes(UTF8),
            "Scottsdale".getBytes(UTF8), clock);
        Column cambriaStateCol = new Column("state".getBytes(UTF8),
            "AZ".getBytes("UTF-8"), clock);
        Column cambriaZipCol = new Column("zip".getBytes(UTF8),
            "85255".getBytes(UTF8), clock);
```

```
        ColumnOrSuperColumn nameCosc = new ColumnOrSuperColumn();
        nameCosc.column = cambriaNameCol;

        ColumnOrSuperColumn phoneCosc = new ColumnOrSuperColumn();
        phoneCosc.column = cambriaPhoneCol;

        ColumnOrSuperColumn addressCosc = new ColumnOrSuperColumn();
        addressCosc.column = cambriaAddressCol;

        ColumnOrSuperColumn cityCosc = new ColumnOrSuperColumn();
        cityCosc.column = cambriaCityCol;

        ColumnOrSuperColumn stateCosc = new ColumnOrSuperColumn();
        stateCosc.column = cambriaStateCol;

        ColumnOrSuperColumn zipCosc = new ColumnOrSuperColumn();
        zipCosc.column = cambriaZipCol;

        Mutation nameMut = new Mutation();
        nameMut.column_or_supercolumn = nameCosc;
        Mutation phoneMut = new Mutation();
        phoneMut.column_or_supercolumn = phoneCosc;
        Mutation addressMut = new Mutation();
        addressMut.column_or_supercolumn = addressCosc;
        Mutation cityMut = new Mutation();
        cityMut.column_or_supercolumn = cityCosc;
        Mutation stateMut = new Mutation();
        stateMut.column_or_supercolumn = stateCosc;
        Mutation zipMut = new Mutation();
        zipMut.column_or_supercolumn = zipCosc;

        //设置 batch
        Map<byte[], Map<String, List<Mutation>>> cambriaMutationMap =
            new HashMap<byte[], Map<String, List<Mutation>>>();

        Map<String, List<Mutation>> cambriaMuts =
            new HashMap<String, List<Mutation>>();
        List<Mutation> cambriaCols = new ArrayList<Mutation>();
        cambriaCols.add(nameMut);
        cambriaCols.add(phoneMut);
        cambriaCols.add(addressMut);
        cambriaCols.add(cityMut);
        cambriaCols.add(stateMut);
        cambriaCols.add(zipMut);

        cambriaMuts.put(columnFamily, cambriaCols);

        //外层映射是行键值
        //内层映射是列族名
        cambriaMutationMap.put(cambriaKey.getBytes(), cambriaMuts);
        return cambriaMutationMap;
    }
}
```

这个例子有点长，给出了比“Hello, world”更多的信息——例子中包含很多普通列族和超级列族的 insert 和 batch_mutate 操作。其中，我刻意选择了多行不同类型的数据，这样会让查询也复杂一些。

这个应用会首先运行这个类，一旦预装载方法执行完成，数据库里就会拥有所有这些数据，从而可以用这些数据进行搜索了。

4.4.5 搜索应用

例 4-7 是可供执行的带有 main 方法的 Java 类。它依赖于 Log4J 来输出信息，所以，在运行前可以先设置 log4j.properties 文件。你需要做的所有事情就是运行这个类，随后数据库就会预装入所有的酒店和兴趣点信息。之后，程序让用户查找指定城市中的酒店。用户选择一个酒店后，程序会取出附近的兴趣点。如果你有兴趣，可以在这个程序的基础上，继续实现应用的其他部分，来提供预定房间的功能。

例 4-7：HotelApp.java

```
package com.cassandraguide.hotel;

import static com.cassandraguide.hotel.Constants.CL;
import static com.cassandraguide.hotel.Constants.UTF8;

import java.util.ArrayList;
import java.util.List;

import org.apache.cassandra.thrift.Cassandra;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.KeyRange;
import org.apache.cassandra.thrift.KeySlice;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.cassandra.thrift.SuperColumn;
import org.apache.log4j.Logger;

/**
 * 运行酒店应用。在数据库预装载后，这个类会模拟一个用户交互行为，
 * 搜索一个城市中的酒店，选择其中一个，然后查看酒店周围的兴趣点。
 *
 * 这个类中会展示具体化视图模式、get、get_range_slice、key slices
 *
 * 为了简化代码，main 方法会直接抛出下列异常：
 * UnsupportedEncodingException,
   InvalidRequestException, UnavailableException, TimedOutException,
   TException, NotFoundException, InterruptedException

   对于一些常用字符串，使用了 Constants 类。
 */
```

```
public class HotelApp {
    private static final Logger LOG = Logger.getLogger(HotelApp.class);

    public static void main(String[] args) throws Exception {

        //先把所有数据放入数据库
        new Prepopulate().prepopulate();
        LOG.debug("** Database filled. **");

        //现在运行客户端
        LOG.debug("** Starting hotel reservation app. **");
        HotelApp app = new HotelApp();

        //通过城市来查找酒店，尝试 Scottsdale 或是纽约
        List<Hotel> hotels = app.findHotelByCity("Scottsdale", "AZ");
        //List<Hotel> hotels = app.findHotelByCity("New York", "NY");
        LOG.debug("Found hotels in city. Results: " + hotels.size());

        //选择一个
        Hotel h = hotels.get(0);

        LOG.debug("You picked " + h.name);

        //查找酒店周边的兴趣点
        LOG.debug("Finding Points of Interest near " + h.name);
        List<POI> points = app.findPOIByHotel(h.name);

        //选择一个兴趣点
        POI poi = points.get(0);
        LOG.debug("Hm... " + poi.name + ". " + poi.desc + "--Sounds fun!");
        LOG.debug("Now to book a room...");

        //显示有空房的日期
        //留为练习 ...
        //预定房间
        //留作练习 ...
        LOG.debug("All done.");
    }

    //使用列分片来读取超级列
    public List<POI> findPOIByHotel(String hotel) throws Exception {

        ///查询
        SlicePredicate predicate = new SlicePredicate();
        SliceRange sliceRange = new SliceRange();
        sliceRange.setStart(hotel.getBytes());
        sliceRange.setFinish(hotel.getBytes());
        predicate.setSlice_range(sliceRange);

        // 读取这行的所有列
        String scFamily = "PointOfInterest";
        ColumnParent parent = new ColumnParent(scFamily);

        KeyRange keyRange = new KeyRange();
        keyRange.start_key = "".getBytes();
        keyRange.end_key = "".getBytes();
```

```
        List<POI> pois = new ArrayList<POI>();

        //取出的不是一个简单的列表，而是一个映射，映射的键值是行键值
        //值是行键值对应的列的列表
        //只有行键值+第一列是有索引的
        Connector cl = new Connector();
        Cassandra.Client client = cl.connect();
        List<KeySlice> slices = client.get_range_slices(
            parent, predicate, keyRange, CL);

        for (KeySlice slice : slices) {
            List<ColumnOrSuperColumn> cols = slice.columns;

            POI poi = new POI();
            poi.name = new String(slice.key);

            for (ColumnOrSuperColumn cosc : cols) {
                SuperColumn sc = cosc.super_column;

                List<Column> colsInSc = sc.columns;

                for (Column c : colsInSc) {
                    String colName = new String(c.name, UTF8);
                    if (colName.equals("desc")) {
                        poi.desc = new String(c.value, UTF8);
                    }
                    if (colName.equals("phone")) {
                        poi.phone = new String(c.value, UTF8);
                    }
                }

                LOG.debug("Found something neat nearby: " + poi.name +
                    ". \nDesc: " + poi.desc +
                    ". \nPhone: " + poi.phone);
                pois.add(poi);
            }
        }

        cl.close();
        return pois;
}

    //使用键值区间
    public List<Hotel> findHotelByCity(String city, String state)
        throws Exception {

        LOG.debug("Seaching for hotels in " + city + ", " + state);

        String key = city + ":" + state.toUpperCase();

        ///查询
        SlicePredicate predicate = new SlicePredicate();
        SliceRange sliceRange = new SliceRange();
        sliceRange.setStart(new byte[0]);
        sliceRange.setFinish(new byte[0]);
        predicate.setSlice_range(sliceRange);
```

```
        // 读取行中的所有列
        String columnFamily = "HotelByCity";
        ColumnParent parent = new ColumnParent(columnFamily);

        KeyRange keyRange = new KeyRange();
        keyRange.setStart_key(key.getBytes());
        keyRange.setEnd_key((key+1).getBytes()); // 键值区间外一点
        keyRange.count = 5;

        Connector cl = new Connector();
        Cassandra.Client client = cl.connect();
        List<KeySlice> keySlices =
            client.get_range_slices(parent, predicate, keyRange, CL);

        List<Hotel> results = new ArrayList<Hotel>();

        for (KeySlice ks : keySlices) {
            List<ColumnOrSuperColumn> coscs = ks.columns;
            LOG.debug(new String("Using key " + ks.key));

            for (ColumnOrSuperColumn cs : coscs) {

                Hotel hotel = new Hotel();
                hotel.name = new String(cs.column.name, UTF8);
                hotel.city = city;
                hotel.state = state;

                results.add(hotel);
                LOG.debug("Found hotel result for " + hotel.name);
            }
        }

        // 查询结束
        cl.close();

        return results;
    }
}
```

我在代码中插入了零散的注释，以解释每段代码的用途。

程序运行的输出如例 4-8 所示。

例 4-8：酒店应用运行的输出

```
DEBUG 09:49:50,858 Inserted AZC_043
DEBUG 09:49:50,861 Inserted AZS_011
DEBUG 09:49:50,863 Inserted CAS_021
DEBUG 09:49:50,864 Inserted NYN_042
DEBUG 09:49:50,864 Done inserting at 6902368219815217
DEBUG 09:49:50,873 Inserted HotelByCity index for Cambria Suites Hayden
DEBUG 09:49:50,874 Inserted HotelByCity index for Clarion Scottsdale Peak
DEBUG 09:49:50,875 Inserted HotelByCity index for The W SF
DEBUG 09:49:50,877 Inserted HotelByCity index for The Waldorf=Astoria
DEBUG 09:49:50,877 Inserting POIs.
DEBUG 09:49:50,880 Done inserting Empire State.
```

```
DEBUG 09:49:50,881 Done inserting Central Park.
DEBUG 09:49:50,885 Done inserting Phoenix Zoo.
DEBUG 09:49:50,887 Done inserting Spring Training.
DEBUG 09:49:50,887 Done inserting POIs.
DEBUG 09:49:50,887 ** Database filled. **
DEBUG 09:49:50,889 ** Starting hotel reservation app. **
DEBUG 09:49:50,889 Seaching for hotels in Scottsdale, AZ
DEBUG 09:49:50,902 Using key [B@15e9756
DEBUG 09:49:50,903 Found hotel result for Cambria Suites Hayden
DEBUG 09:49:50,903 Found hotel result for Clarion Scottsdale Peak
DEBUG 09:49:50,904 Found hotels in city. Results: 2
DEBUG 09:49:50,904 You picked Cambria Suites Hayden
DEBUG 09:49:50,904 Finding Points of Interest near Cambria Suites Hayden
DEBUG 09:49:50,911 Found something neat nearby: Phoenix Zoo.
Desc: They have animals here..
Phone: 480-555-9999
DEBUG 09:49:50,911 Found something neat nearby: Spring Training.
Desc: Fun for baseball fans..
Phone: 623-333-3333
DEBUG 09:49:50,911 Hm... Phoenix Zoo. They have animals here.--Sounds fun!
DEBUG 09:49:50,911 Now to book a room...
DEBUG 09:49:50,912 All done.
```

提示一下，通常不需要直接写 Thrift 或是 Avro，而应该使用第 8 章给出的某个客户端。这个例子的目的是解释使用 Cassandra 工作的一些具体情况，并展示一个完整可用的应用是如何进行各种插入、查询的，就像在现实世界使用一样。

4.5 Twissandra

当你开始研究如何用 Cassandra 进行设计的时候，可以先看看 Eric Florenzano 写的 Twissandra。这是一个可以工作的 Twitter 的克隆，在 http://www.twissandra.com 下载并试用。源代码是用 Python 写的，有一些对 Django 和 JSON 库的依赖关系需要解决，不过仍然是个很不错的起点。这里可以使用你非常熟悉的数据模型（如 Twitter 的），看到其中的用户、时间线、消息等是如何放到一个简单的 Cassandra 数据模型当中的。

Eric Evans 有一篇非常不错的博客文章，介绍了如何使用 Twissandra，这篇文章位于 http://www.rackspacecloud.com/blog/2010/05/12/cassandra-by-example[1]。

4.6 小结

本章我们学习了如何创建一个完整可用的 Cassandra 应用。我们对比了一个典型的关系模型，展示了如何在 Cassandra 中完成同样的工作，并且介绍了和数据库交互的多种方法。

译注1：译者曾翻译过此篇文章，位于http://wangxu.me/blog/p/383。

第 5 章

Cassandra 的架构

在本章中，我们将探讨 Cassandra 的内部设计，以便理解它是如何工作的。我们将剖析 Cassandra 的 P2P 设计，以及对应的 gossip 协议，Cassandra 如何处理读写请求，并研究这些设计抉择是如何影响 Cassandra 的架构性考虑的，这些考虑包括可扩展性、持久性、可用性、可管理性等。我们还将讨论 Cassandra 所采用的分阶段事件驱动架构，这种架构在 Cassandra 中是作为请求代理平台的。

Cassandra 的架构非常精巧，构建于多种理论结构之上。在讨论这里的新名词时，可能会不可避免地引用其他还出现过的新名词。这确实有点让人沮丧，因此我在本书的最后加入了一个词汇表，以作参考。

5.1 system keyspace

Cassandra 有一个称为 `system` 的内部 keyspace，用于存储关于集群的原数据，以帮助各种操作顺利进行。在微软的 SQL Server 中，同样维护着两个元数据库：`master` 和 `tempdb`。`master` 用于保存磁盘空间、使用率、系统设置以及一般性的服务安装信息，而 `tempdb` 则是一个工作空间，用于存储中间结果和完成一般性任务的。Oracle 数据库也有一个称为 `SYSTEM` 的表空间，用作类似用途。Cassandra 的 `system` keyspace 的用途与这些数据库非常类似。

特别指出，`system` keyspace 不仅存储了用于本地节点的元数据，也存储提示切换信息。这些元数据包括：

- 节点令牌；
- 集群名；

- 用于支持动态装载的 keyspace 和 schema 的定义；
- 迁移数据；
- 节点是否自举成功。

schema 的定义存储于两个列族之中：`Schema` 列族保存用户的 keyspace 和 schema 的定义，而 `Migrations` 列族则用于记录对 keyspace 的变更。

system keyspace 是无法手工修改的。

5.2 对等结构

在传统的多节点部署的数据库（如 MySQL），甚至是使用较新模型的数据库产品（如 Google 的 Bigtable）中，某些节点会被设计为主节点，其他节点则作为从节点。它们在集群中扮演不同的角色：主节点具有对数据的控制权，从节点与主节点进行数据同步。任何变更都会写入主节点，并从主节点传送给从节点。这个模型是对读数据优化的，因为它允许客户端从任意一个从节点读取信息。但是在这个模型中，数据复制是从主到从单向的。这带来的一个严重后果就是，所有写操作都必须送到主节点；也就是说，存在一个潜在的单点故障。在主 / 从设置情况下，主节点掉线的后果会很严重。

Cassandra 与此相反，采用了对等结构（P2P）的分布式模型。在这个模型中，从结构上说，所有节点的地位都彼此相同，也就是说，没有主从节点的差别。Cassandra 的设计目标就是整个系统的可用性和可扩展性。P2P 设计可以增强整个数据库的可用性，因为虽然任意 Cassandra 节点掉线都可能会影响系统的整体吞吐能力，但这是一个非常缓和的降质过程，不会中断服务。如果使用了合理的副本复制策略，故障节点上的所有数据将仍然可以被读写。

这种设计还让 Cassandra 更加易于通过增加节点来扩展系统。因为所有节点的行为是相同的，要增加新的服务器，只要简单地把节点加入集群就可以了。新节点不会立刻开始接受请求，它会有一定时间用于学习整个环的拓扑，并接收它可能将要负责的数据。在这之后，它就可以加入这个环并开始接受请求了。这是一个非常自动化的过程，只需要极少的配置工作。因此，相比于主 / 从副本复制模式，P2P 设计让规模增长和缩减都更加容易。

5.3 gossip与故障检测

为了做到无中心、容忍网络分裂，Cassandra 使用了一个 gossip（流言）协议来进行环内通信，这样每个节点都会有其他节点的状态信息。gossiper（流言者）在定时器

的控制之下，每秒钟运行一次。提示移交（hinted handoff）是由 gossip 触发的，当一个节点发现另一个节点重新在线，而它正好有关于这个节点的提示信息时，就会触发提示移交。与提示移交不同，逆熵是一个手动过程，不是由 gossip 触发的。

gossip 协议（流言协议，有时也叫做“传染协议”），通常假设网络是不可靠的，常见于大规模、无中心的网络系统，经常作为分布式数据库中的一种自动数据副本复制机制。gossip 得名于流言传播（gossip）的概念，是一种节点可以按照自己的期望，自行选择与之交换信息的节点的通信方式。



“gossip 协议”这个名词是 1987 年由当时施乐公司帕洛阿尔托研究中心的研究员 Alan Demer 发明的，他当时正在研究不可靠网络中的路由信息传播方法。

Cassandra 中的 gossip 协议主要是在 `org.apache.cassandra.gms.Gossiper` 类中实现的，这个类用于管理本地节点的 gossip 通信。当一个服务节点启动后，它会把自己注册到 gossiper 上，来接收端点状态信息。

因为 Cassandra 的 gossip 会用于故障检测，所以 `Gossiper` 类会维护一个节点列表，存储节点的死活信息。

gossiper 是这样工作的。

(1) G=gossiper（按照 `TimerTask` 的设置）周期性地运行，在环里随机选择一个节点，发起对这个节点的 gossip 会话。每轮 gossip 需要三条消息。
(2) gossip 的发起者给它选好的伙伴节点发送一条 `GossipDigestSynMessage`。
(3) 当伙伴节点收到消息时，回复一条 `GossipDigestAckMessage`。
(4) 发起者收到伙伴发回的响应消息后，再向伙伴发送一条 `GossipDigestAck2-Message`，以此完成本轮 gossip。

当 gossiper 发现一个端点已经死亡的时候，就会通过在它本地的列表中将这个节点标记为死亡来对这个节点“宣判”，并会记入日志。

Cassandra 的故障检测非常强健，使用了一个在分布式计算领域中非常流行的 Phi 增量故障检测算法。这种故障检测机制是在 2004 年由日本先进科学技术研究所首先提出的。

增量故障检测基于两个基本思路。第一个整体思路是，故障检测应该是灵活的，这通过将算法与被监测的应用解耦来实现。而第二个思路，是一个有别于传统故障检测的更新颖的思路，传统方法使用简单的“心跳”机制来判断节点是否已死——通过能否收到心跳判断节点是否已死。但是增量故障检测认为这种方法过于幼稚，与此不同，它们会在死活两个极端之间来寻找一个中间的位置——嫌疑级别。

这样，故障监测系统会根据对节点发生故障的确信程度，输出一个连续变化的“嫌疑”级别，这种方法的优点是，它将网络环境的波动性考虑在内了。例如，只因为丢失一个连接并不能确定一个节点已经死了。嫌疑级别会基于观测（心跳的采样）来得出一个更加连续的、具有前摄性的指示，判断或强或弱的故障可能性，而不是一个简单的死活断言。



可以在这里 http://ddg.jaist.ac.jp/pub/HDY+04.pdf 阅读 Naohiro Hayashibara 等人关于 Phi 增量故障检测的论文。

Cassandra 中的故障检测在 `org.apache.cassandra.gms.FailureDetector` 类中实现，这个类实现了 `org.apache.cassandra.gms.IFailureDetector` 接口。它们共同允许进行如下操作。

- `isAlive(InetAddress)`
  故障检测器会返回一个节点的存活度报告。

- `interpret(InetAddress)`
  由 gossiper 使用，基于通过计算 Phi 得到的嫌疑级别，帮助 gossiper 确定一个节点是否存活（Phi 的计算如 Hayashibara 的论文所述）。

- `report(InetAddress)`
  当一个节点接收到一个心跳时，会调用这个方法。

5.4 逆熵与读修复

在谈到 gossip 协议的地方，你常常还会发现一个与之对应的机制——逆熵，这也是一种基于传染病理论的算法。*逆熵*（anti-entropy）是 Cassandra 的副本同步机制，用于保障不同节点上的数据都更新到最新的版本。

接下来就是逆熵是如何工作的。服务器在主压紧操作期间，会与邻居节点进行一个 TreeRequest/TreeReponse 会话，交换 Merkle 树。Merkle 树是列族数据的一个哈希表示。如果两个节点的树不匹配，它们就必须进行协商（或是“修复”），以确保两者都持有最新的数据。树比较确认机制是 `org.apache.cassandra.service.AntiEntropyService` 类的职责。`AntiEntropyService` 使用了 Singleton 模式，并定义了静态的 `Differencer` 类，可以用于比较两棵树。如果它在两棵树之间发现了任何差异，都会对差异部分启动一个修复过程。

亚马逊的 Dynamo 中使用了逆熵，Cassandra 的实现就使用的 Dynamo 的模型（参考 Dynamo 论文的 4.7 节）。

Dynamo 在逆熵中使用了 Merkle 树（参见词汇表中 Merkle 树的定义）。Cassandra 也使用了 Merkle 树，但两者的实现略有不同。在 Cassandra 中，每个列族都有自己的 Merkle 树，在主压紧操作过程中（参见词汇表中的“压紧”），Merkle 是作为一个快照被创建的，生命周期仅限于它被需要发送给环上的邻居节点的时候。这样做的优点是降低了磁盘 I/O。

在每次更新之后，逆熵算法都被引入。这会对数据库进行校验和，并与其他节点比较校验和。如果校验和不同，就进行数据交换。这需要一个时间窗口来保证其他节点可以有机会得到最近的更新，这样系统就不会总是进行没有必要的逆熵操作。为了保证这个操作非常快，节点内部需要保留一个基于时间戳的反向索引，只交换最近的更新。

在 Cassandra 中，集群由多个节点组成，其中的一个或多个会作为某块给定数据的副本。要读取数据，客户端连接到任意集群中的任意一个节点即可，基于用户指定的一致性级别，一定数量的节点会被读取。在客户端指定的一致性级别没有达到之前，读操作是阻塞的。如果 Cassandra 检测到某些响应节点持有的是过时数据，它会向用户返回最新的数据。在返回之后，Cassandra 会在后台进行一个读修复过程。这个操作会更新过时的数据。

这个设计不仅在 Cassandra 存在，很多键 / 值存储系统之中也是如此，如 Voldemort 项目或是 Riak 项目。这被认为是一个性能改进，因为客户端不必一直阻塞到所有节点都被读取的时候，读修复阶段的更新数据任务会在后台执行。如果你有很多客户端，就很有必要从特定一组节点读取，以确保至少一个节点会有最新的值。

如果客户端指定了弱一致性级别（比如 `ONE`），那么读修复会在返回给客户端之后的后台进行。如果使用了两种更强的一致性级别之一（`QUORUM` 或 `ALL`），那么读修复在数据返回给客户端之前就进行了。

如果一个读操作发现了同一时间戳的不同值，Cassandra 会直接使用一个决胜机制来进行值的比较，以确保读修复不会进入死循环。这种情况应该极少出现。

5.5 memtable、SSTable和commit log

进行写操作的时候，数据是直接写入到 commit log 中的。commit log 是 Cassandra 为了达到持久性而引入的一种错误恢复机制。写操作只有写入到 commit log 才被认为是成功的，这样，即使数据还没有进入内存存储结构中（马上就要介绍到的 memtable），也可以进行数据恢复。

数据写入到 commit log 中之后，会写入到称为 memtable 的内存数据结构之中。当

memtable 之中存储的对象数量达到阈值之后，memtable 会被刷入磁盘，放在一个称为 SSTable 的文件中。然后，创建一个新的 memtable，接收数据。这个刷写的过程是个非阻塞操作，对于一个列族，可以有多个 memtable，一个是当前的，其他的则是等待写入磁盘的。这个过程通常不会很长，除非节点过载了，否则刷写的过程应该会很快。

每个 commit log 都有一个内部的标志位，用于标识其是否需要刷写。当接收到一个写操作时，写的内容被写入到 commit log 之中，并置标志位为 `1`。每个列族只有一个标志位，因为一台服务器上，只有一个 commit log 最终会被写入。对所有列族的全部写操作都会进到同一个 commit log 之中来，所以这个标志位用于标识某个 commit log 对于某个特定的列族是否还包含没有被写入的东西。一旦 memtable 被正确刷入到磁盘中去，对应的 commit log 的标志位就会置回为 `0`，这就表示不存在需要持久化的数据了。和一般的日志文件一样，commit log 也有一个可配置的处理阈值，一旦这个文件达到阈值了，日志会被翻转，所有现存已使用的标志位都会被写入到磁盘中去。

SSTable 的概念是从 Google 的 Bigtable 里借鉴过来的。一旦 memtable 被刷写入磁盘，成为一个 SSTable，它就是不可变的了，不能被任何应用所修改。尽管 SSTable 是被压紧的，但压紧操作只是改变数据在磁盘上的表现形式。实际上，压紧是进行了归并排序的“归并”步骤，将数据并入新的文件，并删除旧文件。



SSTable 是有序字符串表（Sorted String Table）的缩写，对于 Cassandra 的具体实现来说，这有点名不符实，因为 Cassandra 的数据不是以字符串形式存储的。

每个 SSTable 有一个关联的 Bloom filter，用以提高性能（参见 5.8 节）。

所有的写操作是顺序进行的，这正是 Cassandra 的写操作性能出众的原因。在 Cassandra 之中，写一个值不需要任何读或者定位操作，因为所有的写都是以追加方式写入的。这会对磁盘速度性能有关键的限制。压紧操作定期地重新组织数据，不过压紧操作同样也是进行顺序读写。所以，通过拆分，我们获得了很多性能提升，写操作是直接的追加写，之后的压紧可以组织数据，从而获得更好的读性能。如果 Cassandra 只是简单地直接在最终的位置上写入数据，那会让客户端在写操作时为来回寻道而付出沉重代价。

对于读操作，Cassandra 会首先检查 memtable 来查找值，memtable 是由 `org.apache.cassandra.db.Memtable` 类实现的。

5.6 提示移交

考虑如下场景。一个写请求到达 Cassandra，但是负责这部分数据的节点却由于网络分裂、硬件故障或其他原因而不可用。为了保证整个环在这种情况下的可用性，Cassandra 实现了一个称为*提示移交*（hinted handoff）的机制。可以把一个*提示*看做是一个小即时贴，上面记录着写请求的内容。如果写操作所属的节点失败了，Cassandra 接收到该请求的节点会创建一个提示，包含这样一条备忘信息："我有一个给节点 B 的写请求信息。这个请求现在挂起了，等到节点 B 回来的时候请告知我，那时我会把写请求送交给它。"也就是说，写操作的提示信息将会从节点 A 移交给节点 B。

提示移交允许 Cassandra 对于写操作永远可用，降低离线节点恢复服务之后的不一致的时间。之前我们讨论过一致性级别，你可能还记得，我们曾经提到过 0.6 版本中引入的一致性级别 ANY，这个一致性级别意味着有一个提示移交就可以认为写操作是成功的。也就是说，即使只有一个提示被记录下来，写操作也就可以认为是成功了。

一些对提示移交的顾虑，在 Cassandra 社区内部就已经提出过了。起先，这似乎是一个深思熟虑且精巧的设计，可以保证数据库的持久性，并且，因为这种方法已经在很多分布式计算模式中出现过，比如 Java 消息服务（JMS），似乎不会有什么问题。在具有持久性的"保障传递"JMS 队列中，如果消息无法发送给接收者，JMS 会等待一个给定时间，然后重传消息，直到消息被成功接收。但是在实际系统中，不论是对于 JMS 的可靠传输还是对于 Cassandra 的提示移交，都存在一个问题：如果节点离线持续一段时间，其他节点上会堆积相当多的提示信息。之后当其他节点发现掉线节点重新在线的时候，请求会如潮水般涌向这个节点，而此时，这个节点本身正处在自己最脆弱的状态（它刚刚从故障中恢复过来，正在努力恢复工作）。

作为对顾虑的回应，现在可以完全关闭提示移交，或者，用一个不那么极端的方法，降低提示移交消息相对于新的写请求的优先级。



在 Cassandra 0.6 和更早的版本中，`HintedHandoffManager.sendMessage` 会把一整行读入到内存之中，然后把整行信息在一条消息中返回给客户端。而在 0.7 之中，Cassandra 会给被提示的行进行分页。对于很宽的行，这会带来很多性能改善。

5.7 压紧

在 Cassandra 中，压紧操作用于合并 SSTable。在压紧操作过程中，SSTable 中的数据会被合并：键值进行合并，列被组合在一起，丢弃墓碑，创建新的索引。

压紧操作是通过合并大的累积文件而释放空间的过程。大致类似于关系型数据库里的重建表。不过，正如 Stu Hood 所指出的，它在 Cassandra 中主要的不同在于，这个操作是完全透明的，并且在整个服务器的生命周期中持续进行。

在压紧操作中，合并后的数据是有序的，对这些有序的数据会创建一个新的索引文件，同时上述这些刚刚合并的、有序的、有索引的数据会被写入一个单独的新的 SSTable 之中（每个 SSTable 包含三个文件：数据、索引和过滤程序）。这个过程由 `org.apache.cassandra.db.CompactionManager` 类来管理，`CompactionManager` 实现了一个 MBean 接口，支持内省机制。

压紧的另一个重要功能是通过降低定位的次数来提高性能。对于一个给定的键值，要查找一列数据需要的查找次数的上限由 SSTable 的个数决定。如果一个键值经常改动，那么可能每个改动都会刷写入 SSTable。压紧这些 SSTable 可以避免在查找数据时被迫在每个 SSTable 都查找一次数据。

Cassandra 中有多种不同的压紧操作。*主压紧*的出发原因有两种：通过节点探测触发或是自动进行。节点探测会给被探测节点的相邻节点发送一个 TreeRequest 消息。当一个节点收到 TreeRequest 时，会立刻进行一次*只读压紧*，来验证列族。

只读压紧包含如下步骤。

(1) 获取列族中的键值分布。
(2) 行被加入到验证器中之后，如果列族需要验证，就会创建 Merkle 树，并广播到周边节点。
(3) Merkle 树们被放在一起，作为一个 `Differencers`（需要验证或比较的树）的列表发送。
(4) 比较过程由 `StageManager` 类进行，这个类负责管理执行任务时的并发问题。在压紧时，`StageManager` 使用一个逆熵阶段。它使用 `org.apache.cassandra.concurrent.JMXEnabledThreadPoolExecutor` 类，在一个单线程内执行压紧程序，并使这个操作可以作为一个 MBean，支持内省机制。

你可以通过降低压紧线程的优先级来提高整体性能。这可以使用如下命令行参数来设置：

```
-Dcassandra.compaction.priority=1
```

当然，这会影响 CPU 的使用率，而非 IO 的。

5.8 Bloom filter

Bloom filter 是一种提升性能的手段，得名于其发明者 Burton Bloom。Bloom filter 是一种用于判断一个元素是否是一个集合的成员的超快速、但不确定的算法。称其为不确定性方法是因为 Bloom filter 可能会得到一个“假阳性”结果，但是它不会得到“假阴性”的结果，也就是说判断为属于不一定确实属于，但判断为不属于则一定不属于。Bloom filter 将数据集里的值映射为一个位数组，并将一个大数据集凝炼为一个摘要字符串。按照定义，摘要字符串会占用远少于原始数据的内存空间。Bloom filter 位于内存之中，这样可以减少查找键值时的磁盘访问，从而改善性能。磁盘访问通常会比内存访问慢很多。所以，Bloom filter 可以看做是一类特殊的缓存。当进行查询时，在访问磁盘之前首先检查 Bloom filter。因为不会有假阴性结果，所以，如果 Bloom filter 显示元素不存在就是真的不存在。但如果 Bloom filter 显示这个元素在集合之中，那就可以进一步去访问磁盘，确认是否存在了。

Nodetool 将会添加一项新的 JMX MBean 特性，允许你查看 Bloom filter 返回了多少次假阳性结果，这个操作称为 `getBloomFilterFalsePositives`。



Apache Hadoop、Google Bigtable 和 Squid 缓存服务器也使用了 Bloom filter 了。

5.9 墓碑

你可能了解关系型世界中的“软删除”这个概念。软删除是指，应用并不直接执行 SQL 的 delete 语句，而是使用一个 update 语句，把某列的值变为“已删除”之类的内容。有时，程序员会使用这种方法来支持审计等用途。

在 Cassandra 之中，有个与此类似的概念，称为*墓碑*。这就是所有删除操作的做法，因而也是自动为你执行的。当你执行一个删除操作时，数据并不会被立刻删除。相反，会被视为一个更新操作，在相应的值上放一个墓碑。墓碑是一个删除标记，当执行压紧时，比墓碑更老的内容都会被清理掉。

有一个相关的设置，称为 Garbage Collection Grace Seconds（垃圾回收时延）。这个时间是服务器对一个墓碑进行垃圾回收之前的等待的时间。这个时间默认是 864000 秒，也就是 10 天。Cassandra 会一直跟踪墓碑的年龄，一旦某个墓碑的寿命比 `GCGraceSeconds` 更长了，就会回收它。这个时延的设计目的是留下足够长的时间，以便于恢复数据，如果一个节点宕机超过这个时间，那么它也会被认为是发生故障了，应该被替换掉。

在 Cassandra 0.7 中，这个设置是对每个列族都可配置的（曾经是整个 keyspace 的一个设置项）。

5.10 分阶段事件驱动架构

Cassandra 实现了一个分阶段事件驱动架构（SEDA）。SEDA 是一种为高并发互联网服务设计的通用架构，由 Matt Welsh、David Culler 和 Eric Brewer（就是我们之前提到的 CAP 理论的提出者）在 2001 年的一篇题为“SEDA：一种良态、可伸缩的互联网服务架构”的论文中提出。



原始论文可以在 http://www.eecs.harvard.edu/~mdw/proj/seda 得到。

在一个典型的应用中，一个单独的任务单位经常会在一个线程内来完成。比如一个写操作，会在一个线程里，有始有终。但在 Cassandra 中却有所不同：它的并发模型是基于 SEDA 的，所以，一个工作可以从一个线程开始，之后移交给另一个线程，这个线程还可能会将它再交给下一个线程。但并不是当前线程来决定是否把工作移交给下一个线程。一个操作会细分为不同阶段，与阶段关联的线程池（实际上是 `java.util.concurrent.ExecutorService`）来决定执行的任务。阶段是任务的最基本的单位，一个操作内部可能会有不同阶段之间的状态迁移。因为每个阶段由不同的线程池处理，Cassandra 可以因此获得显著的性能收益。SEDA 设计也意味着 Cassandra 能够更好地管理其内部的资源，因为不同的操作可能都需要磁盘 IO、或者可能是受限于 CPU，抑或是需要网络操作等，不同的线程池可以根据可用资源来管理任务的执行。

一个阶段包含一个输入事件队列、一个事件处理程序和一个相关联的线程池。这些阶段都由一个控制器来控制，控制器负责进行调度和线程的分配。Cassandra 使用 `java.util.concurrent.ExecutorService` 线程池来实现这个并发模型。想要了解具体的实现，可以看 `org.apache.cassandra.concurrent.StageManager` 这个类。

Cassandra 中，作为阶段的操作有：

- 读
- Mutation
- Gossip
- 响应
- 逆熵

- 附在均衡
- 迁移
- 流

一些新增加的操作也实现为阶段。一些阶段针对于 memtable 的一些单元操作（在 `ColumnFamilyStore` 类中）。`StorageService` 里的一致性管理器也是一个阶段。

阶段实现了 `IVerbHandler` 接口，支持给定动词的功能。因为 mutation 的概念实现为一个阶段，所以它既可以用作添加操作，也可以用作删除操作。

SEDA 是一个很强大的架构。因为它是事件驱动的，正如其名，它可以很好的应付并发，而且没有什么耦合性。

5.11 管理器与服务

Cassandra 的基本内部控制机制由一组类组成。这里会简单介绍一下这些类，来帮助读者了解这其中比较重要的一些。第一个要谈到的大概就是 `org.apache.cassandra.thrift.CassandraServer` 类。这个类实现了对 Thrift 调用接口的呼叫，代理了大部分对 `org.apache.cassandra.service.StorageProxy` 的查询操作。

5.11.1 Cassandra守护进程

`org.apache.cassandra.service.CassandraDaemon` 接口对应着一个节点上的 Cassandra 服务的整个生命周期。它包含了你能想到的各种典型的生命周期操作：`start`、`stop`、`activate`、`deactivate` 以及 `destroy`。

5.11.2 存储服务

Cassandra 数据库服务对应于 `org.apache.cassandra.service.StorageService` 类。存储服务持有节点的令牌，这表征了节点应该负责的数据范围。

当服务器启动时，会调用这个类的 `initServer` 方法，在这里注册 SEDA 操作管理器、决定服务器的状态（比如自举是否成功、本节点的分区器是什么），并在 JMX 服务器把自己注册为一个 MBean。

5.11.3 消息服务

`org.apache.cassandra.net.MessagingService` 的用途是创建用于消息交换的套接口监听器的，节点的进出消息都会经过这个服务。 `MessagingService.listen`

方法会创建一个线程。每个到达的连接接下来都会被转交到 ExecutorService 线程池，使用 org.apache.cassandra.net.IncomingTcpConnection（派生自 Thread 的类）来解码消息。消息会首先进行验证，之后判断是否是一条流消息。消息流是 Cassandra 用于在节点间传送 SSTable 文件段的优化方法，除此之外的所有其他通信都是序列化的消息。如果是流消息，消息会交给 IncomingStreamReader 来处理，否则就由 MessagingService 的反序列化执行器来处理，它是以一个执行了 Runnable 的任务的形式传递的。因为这个服务密集使用了“阶段”，而且使用 MBean 封装了它维护的线程池，所以，可以通过 JMX 接口来查看很多关于这个服务如何运行的信息（如读操作是否得到支持等）。

5.11.4 提示移交管理器

正如它的名字，org.apache.cassandra.db.HintedHandoffManager 是内部用于管理提示移交的类。它也维护了一个线程池，同样可以通过 JMX 访问 HINTED-HANDOFF-POOL 来监听。

5.12 小结

本章中，我们学习了 Cassandra 结构的主要支柱，包括 gossip、逆熵、增量故障检测，以及如何使用分阶段事件驱动架构来提升性能。我们还看了 Cassandra 内部是如何执行不同的操作的，比如墓碑和读修复。最后，我们介绍了一些主要的类和接口，如果你想深入研究代码，本章也指出了一些关键点。

第 6 章

配置 Cassandra

在本章里，我们来看看如何配置 Cassandra。我们将逐一地创建 keyspace、设置副本数，并使用合适的副本放置策略。

Cassandra 无须配置即可使用，你可以直接下载、解压，然后运行可执行文件，以默认配置启动服务器。

本章中，我们将聚焦 Cassandra 是如何影响集群中的节点的行为的，包括性能和各种元操作，如副本数、分区和 Snitch。性能调优是另一个单独的话题，将在第 11 章单独介绍。



Cassandra 的开发进度很快，会不断地发生变化。这里，我会尽力跟上版本的变化。

6.1 keyspace

在老版本的 Cassandra 中，keyspace 是直接定义在 XML 配置文件中的，但在 0.7 版本里，你可以使用 API 来动态创建 keyspace 和列族。

在 Cassandra 0.6 或更早的版本里，集群和列族的配置位于一个称为 storage-conf.xml 的文件中。之后有一个从 XML 到 YAML 的过渡转换，所以可以看到 storage-conf.xml 和 cassandra.yaml 两个文件的相关资料。0.7 引入了动态加载，所有对 keyspace 和列族定义的创建和变更都可以通过 Thrift API 和命令行接口实现，不必使用配置文件。

从 Cassandra 0.7 开始，可以使用 API 操作来修改 schema 了，和 SQL 的数据定义语言（DDL）的语法很类似，比如 `CREATE TABLE` 或 `ALTER TABLE`。

一旦 schema 加载到 system keyspace（Cassandra 用于存储集群元数据的内部 keyspace）之后，对 schema 所进行的任何变更都必须使用 Thrift 接口进行。这些操作都以“system”为前缀。提醒你，修改 system keyspace 属于影响很大的 schema 修改操作。

- `system_add_keyspace`
  创建一个 keyspace。

- `system_r  ename_keyspace`
  对一个 keyspace 进行快照，然后修改它的名字。这个方法在执行完成之前会一直阻塞住。
- `system_drop_keyspace` 对一个 keyspace
  进行快照之后，完全删除这个 keyspace。

- `system_add_column_family`
  创建一个列族。

- `system_drop_column_family`
  对一个列族进行快照，然后删除这个列族。

- `system_rename_column_family`
  对一个列族进行快照，然后改名。注意，这个操作在完成之前也会一直阻塞住。

例如，使用 0.7 对命令行创建一个新的 keyspace。可以启动如下命令行工具：

```
[default@unknown] connect 127.0.0.1/9160
Connected to: "Test Cluster" on 127.0.0.1/9160
[default@unknown] create keyspace Test1 with replication_factor=0
610d06ed-a8d8-11df-93db-e700f669bcfc
[default@unknown] describe keyspace Test1
Keyspace: Test1
```

`default@unknown` 的写法和 MySQL 很类似，使用鉴权的用户名（如果需要）和当前使用的 keyspace 名作为提示符。然后可以使用 use keyspace 命令来切换不同的 keyspace：

```
use <keyspace> [<username> 'password']
```

现在，我们切换到在命令行上刚刚创建的 keyspace，可以在其中加入列族：

```
[default@Test1] create column family MyCF
4105a82f-ad51-11df-93db-e700f669bcfc
```

在创建 keyspace 或列族的时候，可以使用 `with` 标记来指定其他附加的设置，并使用 `and` 标记来进行更多设置：

```
[default@MyKeyspace] create keyspace NewKs with replication_factor=1
```

其他的可以使用的命令行命令还包括：

```
drop keyspace <keyspace>
drop column family <cf>
rename keyspace <keyspace> <keyspace_new_name>
rename column family <cf> <new_name>
```

这些工作还可以通过 API 调用来进行，如例 6-1 所示。

例 6-1：使用 API 来动态创建 keyspace 和列族

```
package com.cassandraguide.config;

import java.util.ArrayList;
import java.util.List;

import org.apache.cassandra.thrift.Cassandra;
import org.apache.cassandra.thrift.CfDef;
import org.apache.cassandra.thrift.KsDef;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

public class ConfigAPI {

    private static final String HOST = "localhost";
    private static final int PORT = 9160;

    /**
     * 创建新的 keyspace 和列族。
     */
    public static void main(String... args) throws Exception {

        String keyspaceName = "DynamicKeyspace";
        System.out.println("Creating new keyspace: "+ keyspaceName);

        // 创建 Keyspace
        KsDef k = new KsDef();
        k.setName(keyspaceName);
        k.setReplication_factor(1);
        k.setStrategy_class("org.apache.cassandra.locator.RackUnawareSt-
        rategy");

        List<CfDef> cfDefs = new ArrayList<CfDef>();
        k.setCf_defs(cfDefs);

        // 连接服务器
        TTransport tr = new TSocket(HOST, PORT);
        TFramedTransport tf = new TFramedTransport(tr);
        TProtocol proto = new TBinaryProtocol(tf);
        Cassandra.Client client = new Cassandra.Client(proto);
        tr.open();
```

```
            // 添加新的 keyspace
            client.system_add_keyspace(k);
            System.out.println("Added keyspace: "+ keyspaceName);
        }
    }
```

要创建一个 keyspace，所需要做的就是给定名称、指定副本放置策略（会在 6.3 节中介绍）以及设定副本因子，然后可以按照你的设计，创建任意多个列族，仅此而已。现在，已经是一切就绪，可以向新列族 `MyCF` 里插入数据了。



如果你不指定，Cassandra 会提供一个默认的副本策略。这个例子里，我是为了明确起见，所以展示了语法。

6.1.1 创建列族

可以使用命令行或是 API 来创建一个列族。这些是创建列族时可以指定的选项。

- `column_type`
  标准列族（`Standard`）或是超级列族（`Super`）。

- `clock_type`
  唯一的合法值就是 `Timestamp`。

- `comparator`
  合法选项包括 `AsciiType`、`BytesType`、`LexicalUUIDType`、`LongType`、`TimeUUIDType` 和 `UTF8Type`。

- `subcomparator`
  当 `comlumn_type` 是 `Super` 时，子列使用的比较器。合法值的规定和比较器是一样的。

- `reconciler`
  当列的版本发生冲突的时候，进行协调的类的名字。目前唯一合法的值是 `Timestamp`。

- `comment`
  字符串形式的人类可读的任意注释内容。

- `rows_cached`
  要缓存的行数量。

- `preload_row_cache`
  把这个选项设置为 `true` 可以自动加载行缓存。

- `key_cache_size`
  放入缓存中的键值数量。

- `read_repair_chance`
  合法值区间从 0.0 到 1.0。

这里有一个例子：

```
[default@Keyspace1] create column family MyRadCF
    with column_type='Standard' and comparator='UTF8Type' and rows_
    cached=40000 3ae948aa-ae14-11df-a254-e700f669bcfc
```

6.1.2 从0.6迁移到0.7

放在 conf 目录里的 cassandra.yaml 文件是用来替换之前的 storage-conf.xml 的，后者是 0.6 和之前版本的配置文件。不过，YAML 文件仅仅是用来帮助用户把配置文件从 XML 升级到 YAML 的。也可以如之前的例子所示的那样，通过 Thrift API，使用带有 `system_` 前缀的调用来配置 keyspace 和列族。

如果有一个已有的 0.6 版本的 storage-conf.xml 文件，首先要做的就是使用 bin/config-converter 工具，将它转化成 YAML 格式，它会为你生成一个 cassandra.yaml 文件。在 `org.apache.cassandra.service.StorageServiceMBean` 之中，有一个通过 JMX 输出的操作，称为 `loadSchemaFromYAML`，通过这个操作可以强制 Cassandra 重新加载种子节点中的 cassandra.yaml 文件，让 schema 的改动生效。集群中的新节点会在启动时获得 schema 的更新。这里，种子节点并没有什么特殊的。你可以对任何节点运行这个方法（虽然并不推荐对多个节点运行这个方法），所有节点都将会获得 schema 更新，并通过 gossip 来传播 schema 的变化。将定义信息装入 system keyspace 是个一次性操作，不需要在运行这个命令之后再度修改这个 YAML 文件了。后续的操作，将使用 API 或是命令行接口来进行。一旦版本迁移完成，这个操作就被弃用了。

尽管对于测试来说，使用默认设置会很简单，还是让我们来研究一下如何设置副本放置策略、副本因子和 Endpoint Snitch。这些都是 keyspace 内的设置，不同的 keyspace 可以各不相同。

6.2 副本

集群中的节点越多，副本放置策略也就越重要。在 Cassandra 中，*节点*（node）这个名词使用得非常广泛，是指一个运行着 Cassandra 软件的服务器，隶属于包含一个或多个 Cassandra 服务器的环。

每个 Cassandra 节点都是某些东西的一个副本。对于一个给定的键值区间，有些 Cassandra 节点可能没有这个区间的副本。如果副本因子设置为 1，那么写操作将只写到一个节点上。如果这个节点宕机了，那么存储在这个节点上的数据就不可用了。而如果把副本因子设置成 2，那么在每次写操作中，集群中的两个节点将得到数据，它们互为副本。总之，如果副本因子是 N，那么每个节点将会作为 N 个区间的副本，即使 N 是 `1`。

一个 Cassandra 集群，或是说一组主机，通常被认为是一个环，原因就在这里揭晓。每个环上的节点都有个单一、唯一的令牌（token）。每个节点会声明对一个区间的值的所有权，这个范围从它的令牌直到前一个节点的令牌。区间的定义位于 `org.apache.cassandra.dht.Range` 类中。令牌的形式依赖于你所使用的分区器（partitioner，更多关于分区器的信息可以参考 6.5 节）。

Cassandra 中创建一个副本的时候，第一个副本总位于那个拥有所在键值区间令牌的节点上。所有其他副本的分布则基于副本策略的配置，现在我们就来看看。

6.3 副本放置策略

简单地说，对于配置文件来说，要规定一个副本放置策略，只要提供一个 Java 类的名字即可，这个类需要派生自 `org.apache.cassandra.locator.AbstractReplicationStrategy` 类。配置文件中的这个设置用于配置节点选择器的工作。



要确定副本放置的位置，Cassandra 使用了“四人帮”的策略（Strategy）模式。策略是一个以通用抽象类形式出现的框架，允许使用一个算法的不同的实现（不同的策略来完成同样的工作）。每个算法实现都封装在一个单独的类里，实现同一个接口。所以，你也可以在同一个框架之内来提供算法的不同实现，这些算法是可以在运行时替换的。客户端无须考虑实用的策略。最常见的一个策略模式的例子是排序。考虑一个只有一个排序操作的排序策略接口和各种不同的排序算法实现，如快速排序、归并排序等。每种排序算法的实现都各不相同，选择取决于你的需求。

副本放置策略接口提供了 11 个公开方法需要实现，如果你的策略可以直接使用抽象父类中的一些方法实现的话，也可以只覆盖你需要的方法。当然，你不必自己实现这些策略。Cassandra 已经提供了三个现成的实现，可以直接使用：机架感知策略、机架无关策略和数据中心分片策略。

选择合适的策略非常重要，因为策略决定了每个节点负责的键值范围。这意味着你

要决定哪个节点应该接收哪些写操作，这对于不同场景下的效率会带来巨大影响。如果设置集群时把所有写操作都送到两个远隔重洋的数据中心，一个在澳大利亚，另一个在弗吉尼亚，那么你将看到性能会急剧下降。不同的可置换性策略给了你很大的灵活性，你可以根据具体的网络拓扑来进行调优。

第一个副本总会写到拥有这一键值区间令牌的节点，但其他副本的位置则依赖于你使用的副本放置策略。



在我写作本章的时候，这些策略的名字发生了一些变化。0.7 版本中的名字将会是简单策略 `SimpleStrategy`（之前称为机架无关策略 `RackUnawareStrategy`），旧网络拓扑策略 `OldNetworkTopologyStrategy`（之前称为机架感知策略 `RackAwareStrategy`），以及网络拓扑策略 `NetworkTopologyStrategy`（之前称为跨数据中心分片策略 `DatacenterShardStrategy`）。

6.3.1 简单策略

简单策略是机架无关策略的新名字。

配置文件中，默认使用的策略是 `org.apache.cassandra.locator.RackUnawareStrategy`。这个策略仅仅覆盖了抽象父类的 `calculateNaturalEndpoints` 方法。这个策略在一个数据中心内部放置副本，并不感知副本放置的数据中心与机架。这意味着这种实现在理论上会很快，但如果副本放置的下一个节点位于其他机架上的话，也不会比其他策略快。这个策略如图 6-1 所示。

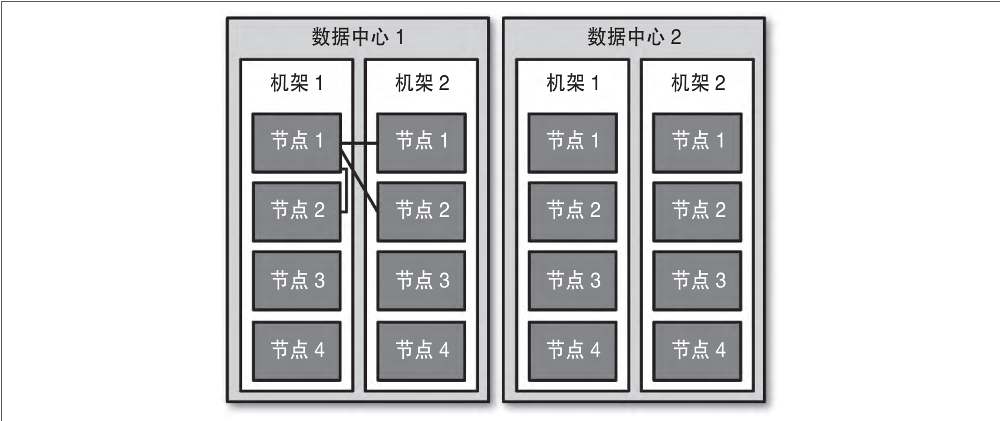


图 6-1：简单策略在一个数据中心内部放置副本，与拓扑无关

这里，实际上是环上的后 N 个节点被选择存放副本，这个策略并不了解数据中心相关信息。图中画出了第二个数据中心，可以看到，这种策略完全没有意识到多个数据中心的存在。

6.3.2 旧网络拓扑策略

Cassandra 提供的第二个可用副本放置策略是 `org.apache.cassandra.locator.RackAwareStrategy`，现在称为旧网络拓扑策略，主要用于将副本分布到同一个数据中心的不同机架之上。和 `RackUnawareStrategy` 一样，这个策略也只覆盖了抽象父类的 `calculateNaturalEndpoints` 方法。这个类如其原始名称所说的那样，可以感知到数据中心中的机架位置。

假设你有 DC1 和 DC2 两个数据中心，里面有一组 Cassandra 服务器。使用这个策略会将一些副本放在 DC1 的不同机架上，还会确保有一个副本放在 DC2。机架感知策略的目标应用场景是，你的 Cassandra 集群节点分布在两个数据中心之中，并且使用副本因子 3。这个策略如图 6-2 所示。

这个策略通过牺牲了一定的时延特性来换取高可用性，因为当与其他数据中心的节点进行通信的时候会带来更高的时延。如果 Cassandra 就在一个数据中心里，没有必要使用机架感知策略。不过，Cassandra 为运行在不同数据中心进行了优化，通过多数据中心获得更高的可用性也许对你是非常重要的。所以如果 Cassandra 分布在多个数据中心，可以考虑使用这个策略。

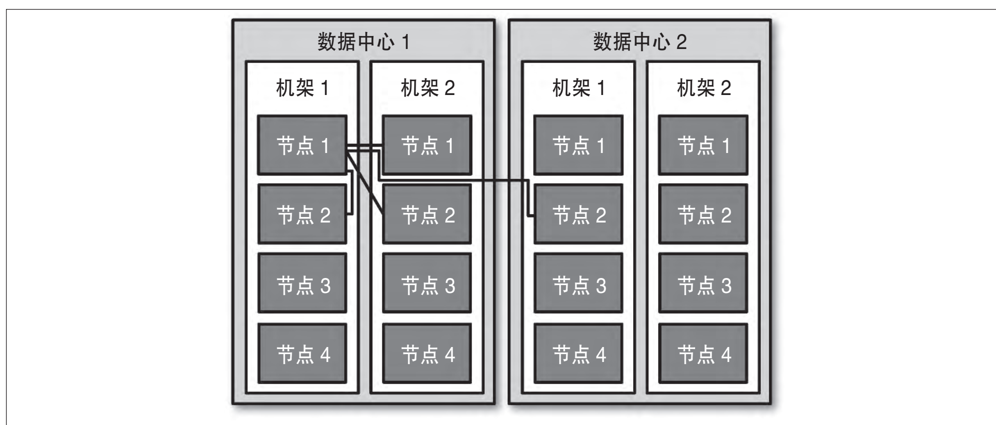


图 6-2：旧网络拓扑策略把第二个副本放在另一个数据中心，然后把其他副本放到本数据中心的不同机架

如果你使用机架感知策略，必须使用机架感知 Snitch。Snitch 在 6.6 节介绍。

6.3.3 网络拓扑策略

相对于机架感知策略（RackAwareStrategy），Cassandra 0.7 中包含的网络拓扑策略允许你更加深度地定制如何将副本分布到不同数据中心。要使用这个策略，需要提供一个参数来指定每个数据中心的副本放置策略。这个文件会被 org.apache.cassandra.locator.DataCenterShardStrategy 类读取并运行，这样，拓扑的设置就更加灵活了。数据中心分片策略如图 6-3 所示。

这个策略曾经使用过一个称为 datacenter.properties 的文件。但在 0.7 当中，这些原数据会直接附加到 keyspace，并做为一个配置选项映射。

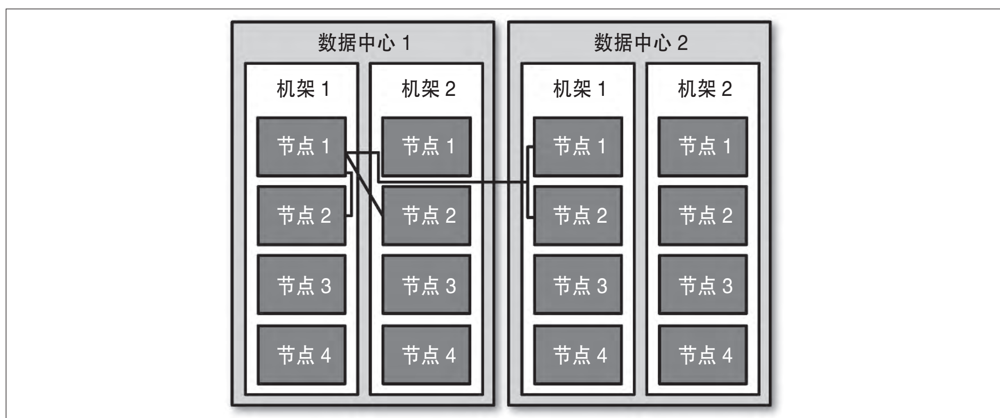


图 6-3：按照用户指定的方式，网络拓扑策略把一些数据放在另一个数据中心，把剩下的副本放在第一个数据中心的其他机架上

6.4 副本因子

副本因子（replication factor）决定了数据将在 Cassandra 集群中存放多少个副本，由 replication_factor 配置项决定。

一个直观的感觉好象是，集群中节点越多，副本因子就应该设置得更高。但是，你不能真的这么做，相反，要根据所需要的服务级别需求来设置副本因子。

当副本因子为 1 的时候，数据在集群中只存放在一个节点上。如果这个节点掉线的话，数据也就不可用了。这也意味着 Cassansra 需要做更多的节点协调工作，如果某个给定键值的所有数据都放在节点 B 上，那么到达节点 A 的每个对于这个键值的客户端请求都需要转发过去。

你也不能设置比节点数更多的副本因子值，因为这没有任何意义。实际上，不应该

通过这个值来调节 Cassandra 的一致性。只要读副本数与写副本数之和大于副本因子，Cassandra 就能够达到高一致性。

所以，如果你有一个 10 个节点的集群，最大能设置的副本因子就是 10，但不应该这么做，这样就让 Cassandra 无从施展自己的长处，也影响了系统的可用性，因为即使有 1 个节点掉线，也无法达到高一致性。相反，正确的用法是将副本因子设为一个合理的值，然后调节一致性级别。这里的“合理的数值”可能会很小。一致性级别不允许将数据写到超过副本因子的节点个数。对于一致性级别，ONE 看起来是最低的，但 ANY 类似于 ONE，却提供更低的一致性，因为你可能会在写入数据之后很久才能看到写入的数据。如果集群中有任何节点还在线，ANY 操作都会成功。



如果你是 Cassandra 的新手，副本因子和一致性级别可能让你觉得有些无所适从。副本因子是 keyspace 的属性，在服务器的配置文件中指定。而一致性级别是每次查询时有客户端指定的。副本因子表示，每次写入操作最终会把数据存入到多少个节点；而一致性级别表示，至少多少个节点响应了，读写操作可以被认为是成功了。有点让人糊涂的是，一致性级别是基于副本因子而定的，而非系统中的节点数。

提升副本因子

从设计上说，副本因子不是一个在运行中调整的参数，而应该在集群启动之前设定。但是，随着应用的增长，你可能需要增加节点，这时可能要提高副本因子。这里有些简单的原则可以在提高副本因子时参考。首先要记住的是，在调高副本因子之后，必须重启节点。修复操作会在节点重启之后进行，Cassandra 将会重新分布数据，以达到副本因子指定的副本数量。在修复的过程中，某些客户端可能会连接到那些还没有得到数据的副本，这样就会收到数据不存在的提示。

把副本因子从 1 提高到 2 的更快的方法是使用 node tool。首先在原始节点执行一个 drain 操作，确保所有数据都刷入 SSTable。之后，停止这个节点，这样它就不会接受新的写操作了。然后复制 keyspace 下的数据文件（配置文件中的 DataFile Directory 配置项的值），确保不要复制内部 Cassandra keyspace 的值。将这些文件粘贴到新节点上。将两个节点上的副本因子配置都提高到 2。确保两个节点上的 autobootstrap 都设置为 false 了。然后重启两个节点，运行 node tool repair。这些操作将让客户端减少忍受未初始化的读操作时间。

为了示意这个过程，我使用了三个节点，IP 地址后缀分别是 1.5、1.7 和 1.8，副本因子为 1。我将连接到节点 1.5，来写入一个之前不存在的列：

```
cassandra> connect 192.168.1.5/9160
Connected to: "TDG Cluster" on 192.168.1.5/9160
cassandra> set Keyspace1.Standard2['mykey']['rf']='1'
Value inserted.
```

现在，我关掉节点 1.5，连接到另一个节点 1.7，来尝试获取我设置的值：

```
cassandra> connect 192.168.1.7/9160
Connected to: "TDG Cluster" on 192.168.1.7/9160
cassandra> get Keyspace1.Standard2['mykey']['rf']
Exception null
```

因为我的副本因子是 1，而这个值仅写入到 1.5 这个节点了，所以当我失去这个节点的时候，集群中的其他节点没有这份数据。在 1.5 节点重新在线之前，客户端将无法读到 `rf` 列的值。

现在我们来看看提高副本因子的效果。把 1.5 和 1.7 节点的副本因子从 1 提高到 2，然后重启。让我们在 1.7 节点上给 `rf` 列插入新值：

```
cassandra> connect 192.168.1.7/9160
Connected to: "TDG Cluster" on 192.168.1.7/9160
cassandra> get Keyspace1.Standard2['mykey']['rf']
Exception null
cassandra> set Keyspace1.Standard2['mykey']['rf']='2'
Value inserted.
```

之前的例子的空响应显示，之前节点 1.7 原本没这份数据。现在关闭 1.7，连接到 1.5，我们看到这个值是因为写到 1.7 的值被复制到了 1.5，这是副本因子的影响：

```
cassandra> connect 192.168.1.5/9160
Connected to: "TDG Cluster" on 192.168.1.5/9160
cassandra> get Keyspace1.Standard2['mykey']['rf']
=> (column=rf, value=2, timestamp=1279491483449000)
```

作为一般性的原则，你可以估算写吞吐量为节点数除以副本因子。所以，如果在副本因子为 1 的时候，一个 10 节点的集群的典型吞吐能力大约是 10 000 次写每秒，那么当把副本因子提高到 2，吞吐能力大约会是 5000 次写每秒。

6.5 分区器

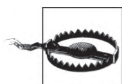
> 未来已经到来，只是分布尚不均匀。
>
> ——William Gibson

分区器（partitioner）的用途是允许你指定行键值应该被如何排序，这会严重影响数据在节点间如何分布，还会影响查询一个范围的行时可选的选项。你可以使用几个不同的分区器，我们会在这里一一介绍。



分区器的选择不适用于列的排序，只用于行键值的排序。

可以通过修改配置文件中的 `Partitioner` 元素或通过 API 来设置分区器。这个元素需要指定一个实现了 `org.apache.cassandra.dht.IPartitioner` 接口的类的名字。Cassandra 自带了三个分区器：默认的随机分区器、有序分区器（order-preserving），以及配页有序分区器（collating order-preserving）。你还可以实现 `org.apache.cassandra.dht.IPartitioner` 接口来创建自己的分区器，并放到 Cassansra 的 classpath 下。



注意，分区器会影响磁盘上的 SSTable 的表示。所以，如果修改了分区器，就不得不删掉数据目录。

6.5.1 随机分区器

随机分区器由 `org.apache.cassandra.dht.RandomPartitioner` 类实现，是 Cassandra 的默认分区器。它使用 `BigIntegerToken` 存放 MD5 哈希值，通过哈希值来决定键值放在环上的具体位置。这样做的好处是可以让键值很均匀地分布到集群中，因为这个分布是随机的。它的不足在于区间查询的效率不高，因为在一个指定区间的键值可能会分布在环上很分散的位置，而且键值的区间查询返回的数据也是随机顺序的。

6.5.2 有序分区器

有序分区器由 `org.apache.cassandra.dht.OrderPreservingPartitioner` 类实现，它实现了 `IPartitioner<StringToken>` 接口。使用这个分区器，令牌是一个基于键值的 UTF-8 字符串。各行是按照键值的顺序存储的，按照排序顺序对齐物理结构。将列族配置为使用有序分区器（OPP），允许你进行区间切片（range slice）操作。

值得注意的是，OPP 在区间查询时并不比随机分区器更有效率——它只是提供顺序性。它的一个缺点是非常容易让环不均衡，因为真实数据通常写得不那么均匀。例如，考虑一下拼字游戏中不同字母的分值。Q 和 Z 很少被用到，所以它们的分值最高。使用 OPP，非常有可能会导致最后大量数据位于某些节点，而其他节点上的数据很少。那些存有很多数据的节点，使得环非常不平衡，常常被看做是“热点”。这样，使用 OPP 就意味着运维团队将不得不周期性地使用 Nodetool 的 `loadbalance` 或 `move` 操作来手工均衡节点。

如果你希望从客户端进行区间查询，就必须使用有序分区器或配页有序分区器。

6.5.3 配页有序分区器

这个分区器使用美国英语区域设置（EN_US）进行键值排序。类似 OPP，这个分区器同样使用 UTF-8 字符串键值。虽然在名字上看，它似乎是扩展了 OPP，实则不然。事实上，它派生自 AbstractByteOrderedPartitioner。因为用途实在有限，这个分区器很少被使用。

6.5.4 字节序分区器

0.7 版本里，开发团队新增加了 ByteOrderedPartitioner，这也是一种有序分区器，它将数据看做是裸字节，而不会像有序分区器和配页有序分区器那样，首先转换为字符串。如果需要有序分区器，但不需要验证键值是否是字符串，推荐使用字节序分区器来提高性能。

6.6 Snitch

Snitch 的工作就是用于确定主机间的大致相互关系。Snitch 收集网络拓扑的信息，以便 Cassandra 可以有效地路由请求信息。Snitch 可以指出某个节点相对于其他节点的位置。它可以为副本放置策略推断节点属于哪个数据中心等信息。

6.6.1 Simple Snitch

Cassandra 默认使用 org.apache.cassandra.locator.EndPointSnitch[1]。它简单地比较节点 IP 地址的每个字节。如果两个主机的地址的第二个字节是一样的，那就认为它们在同一个数据中心，如果两个主机的 IP 地址的第三位是相同的，就认为它们在同一个机架上。“认为是”意味着 Cassandra 的判断基于这样一个假设：你的服务器会放在不同的 VLAN 或子网中。



Simple Snitch 是在 0.7 版本中被重命名的，之前的版本中称为 endpoint snitch。

可以通过修改配置文件中的 <EndPointSnitch> 元素来配置 endpoint snitch。另一个可用的选择是 PropertyFileSnitch。

6.6.2 PropertyFileSnitch

org.apache.cassandra.locator.PropertyFileSnitch 原本位于 contrib 之

译注1：0.7版本中是org.apache.cassandra.locator.SimpleSnitch。

中，在 0.7 版本中被移动到了主代码部分了。这个 Snitch 允许在使用机架感知策略时，通过一个标准的键 / 值 properties 文件来指定更具体的节点位置，配置文件称为 cassandra-rack.properties。

这个 Snitch 由 Digg 开发，它们使用 Cassandra 并经常贡献出它们的开发成果。这个 Snitch 让你告之节点的位置，从而帮助 Cassandra 了解两个 IP 是否位于一个数据中心，或是否在一个机架上。如果因为系统维护需要经常移动服务器，或者采用了一个非常复杂的 IP 地址方案，这个 Snitch 将非常有用。

cassandra-rack.properties 的默认配置是这样的：

```
# Cassandra Node IP=Data Center:Rack
10.0.0.10=DC1:RAC1
10.0.0.11=DC1:RAC1
10.0.0.12=DC1:RAC2

10.20.114.10=DC2:RAC1
10.20.114.11=DC2:RAC1
10.20.114.15=DC2:RAC2

# default for unknown nodes
default=DC1:r1
```

这里可以看到，有两个数据中心，每个有两个机架。Cassandra 可以依此高效地判断出节点的均匀分布。

修改这个文件中的每个值来反映集群的配置，指定有哪些 IP 的节点属于哪个机架、哪个数据中心。虽然如果经常希望增加或移走节点，这个文件会很难维护，但通过牺牲一点灵活性和易维护性换取了更多的控制和更好的运行时效率，因为 Cassandra 不必去判断节点的位置了，而是你直接告诉它每个节点在什么位置。



在晋升为标准发布版的一部分之前，PropertyFileSnitch 曾经叫做 PropertyFileEndPointSnitch，位于 contrib 目录下，如果你希望在网上查找相关信息的话也可以参考原来的名字。

6.7 创建集群

可以在一个单节点上运行 Cassandra，这对于开始熟悉 Cassandra、学习如何读写数据很有用。但 Cassandra 是特别为由很多台机器组成的集群而设计的，可以利用多台计算机的能力，提供很高的容量。在本节中，我们可以看到，要让一个环上的多个 Cassandra 实例相互通信，都需要做什么。



在本书写作时，整个配置机制正在发生翻天覆地的变化。本节适用于如何使用 0.6 版本来创建一个集群。

这里，我们要做的是使用 Cassandra 自带的示例 keyspace 来确保多台机器在同一个环上。我们将从默认配置文件和 keyspace 定义开始，只改变那些用来创建一个简单集群需要的设置。

在这个练习中，假设我们有两台机器，要配置为一个 Cassandra 集群，IP 地址分别为 192.168.1.5 和 192.168.1.7。当启动 Cassandra 时，它读取配置文件来确定你希望当前的节点如何对外广播，也就是说，绑定哪个 IP 地址和端口。你需要告诉当前的节点其他节点的信息，这样它才可以参与到同一个环中。这些都可以通过在文本编辑器里编辑配置文件，修改几个值来完成，正如我们将要给出的例子这样。

集群中的新节点需要一个*种子节点*（seed node）。如果节点 A 作为节点 B 的种子节点，当节点 B 上线时，节点 B 会把节点 A 当做一个参考点，从它获取数据。种子节点会忽略 `AutoBootstrap` 设置，因为它会认为自己是集群中的第一个节点。

6.7.1 修改集群名称

Cassandra 集群有一个名字，这样可以避免一个集群中的节点误加入一个你不希望它参与的集群。配置文件中默认的集群名称为“Test Cluster”。可以通过更新 `<clusterName>` 元素来修改集群名称——要确定已经在希望加入到集群中的所有节点上都修改了这个值了。



如果已经在现有的 Cassandra 集群中写入了数据，然后再修改集群名称，Cassandra 会在尝试启动节点读取数据文件时给出集群名称不匹配的警告，然后自己关闭。

6.7.2 给集群增加节点

配置文件中有个元素默认设为了 false。假设你已经运行了一个集群，或者有一个节点并往里加入了数据，现在希望在集群中加入更多节点，可以使用 `autobootstrap` 元素来做到这点。我有一个 Cassandra 服务器，运行在 IP 地址后缀为 1.5 的机器上，下面是 NodeTool 的输出：

```
$ bin/nodetool -h 192.168.1.5 ring
Address       Status   Load        Range                                   Ring
192.168.1.5   Up       433.43 MB   126804671661649450065809810549633334036 |<--|
```

作为一个种子节点，这台服务器有自己的 IP 地址，我们设 autobootstrap 为 false，因为这个节点本身是种子。如果要增加新的种子节点，那么需要先让它 autobootstrap，然后将它变为种子节点。如果节点 B 要使用节点 A 作为种子，那么节点 B 就应该在它的配置文件中把节点 A 设为种子节点。但节点 A 不需要自己声明为种子。

现在我要加入另一个节点来分担负载。这个节点的地址后缀为 1.7。首先，确定集群中的所有节点设置的集群名和 keyspace 定义都是相同的，这样，新节点才可以接收数据。编辑第二个节点的配置文件，指定第一个节点作为种子。然后把 autobootstrap 设置为 true。

当第二个节点启动的时候，它会立刻发现第一个节点，之后会休眠 90 秒钟，让节点们通过 gossip 传播信息，确定每个节点应该存储多少数据，之后，从第一个节点获取自举令牌（bootstrap token），这样就知道自己将要接收到的数据了。自举令牌的负载是最高负载节点的一半。之后，第二个节点再次休眠 30 秒，然后开始自举。

```
INFO 11:45:43,652 Starting up server gossip
INFO 11:45:43,886 Joining: getting load information
INFO 11:45:43,901 Sleeping 90000 ms to wait for load information...
INFO 11:45:45,742 Node /192.168.1.5 is now part of the cluster
INFO 11:45:46,818 InetAddress /192.168.1.5 is now UP
INFO 11:45:46,818 Started hinted handoff for endPoint /192.168.1.5
INFO 11:45:46,865 Finished hinted handoff of 0 rows to endpoint /192.
     168.1.5
INFO 11:47:13,913 Joining: getting bootstrap token
INFO 11:47:16,004 New token will be 417076584707468130567131241040911 56
     313 to assume load from /192.168.1.5
INFO 11:47:16,019 Joining: sleeping 30000 ms for pending range setup
INFO 11:47:46,034 Bootstrapping
```

根据所拥有的数据量，你会看到新节点在一定时间之后进入工作状态。可以使用 Nodetool 的 `streams` 命令来观察自举过程中的数据传输。查看日志文件也是确定自举完成的好办法，但要在自举进行过程中观测发生了什么，还是要使用 `nodetool streams`。最终，新节点将会从第一个节点那儿接收到负载，你将得到一个新节点已经启动的操作成功提示：

```
INFO 11:52:29,361 Sampling index for /var/lib/cassandra/data\Keyspace1\
     Standard 1-1-Data.db
INFO 11:52:34,073 Streaming added /var/lib/cassandra/data\Keyspace1\
     Standard1-1-Data.db
INFO 11:52:34,088 Bootstrap/move completed! Now serving reads.
INFO 11:52:34,354 Binding thrift service to /192.168.1.7:9160
INFO 11:52:34,432 Cassandra starting up...
```

如你所见，数据传输大概用了 4 分钟左右。

在自举期间，在 1.5 的（种子）节点看起来是这样的：

```
INFO 11:48:12,955 Sending a stream initiate message to /192.168.1.7...
INFO 11:48:12,955 Waiting for transfer to /192.168.1.7 to complete
INFO 11:52:28,903 Done with transfer to /192.168.1.7
```

现在，我们可以再次运行 `nodetool`，来确定一切设置正常：

```
$ bin/nodetool -h 192.168.1.5 ring
Address       Status  Load       Range                                      Ring
                      126804671661649450065809810549633334036
192.168.1.7   Up      229.56 MB  41707658470746813056713124104091156313    |<--|
192.168.1.5   Up      459.26 MB  126804671661649450065809810549633334036   |-->|
```

Cassandra 已经通过从之前的节点（1.5）分出了一半的负载给 1.7，成功的自举了这个节点。现在我们有了一个两节点的集群。要确保它已经工作了，我们在 1.5 添加一个值：

```
cassandra> connect 192.168.1.5/9160
Connected to: "TDG Cluster" on 192.168.1.5/9160
cassandra> set Keyspace1.Standard2['mykey']['col0']='value0'
Value inserted.
```

然后打开第二个命令行客户端，从 1.7 节点读这个值：

```
cassandra> connect 192.168.1.7/9160
Connected to: "TDG Cluster" on 192.168.1.7/9160
cassandra> get Keyspace1.Standard2['mykey']['col0']
=> (column=col0, value=value0, timestamp=1278878907805000
```

你可以重复这些步骤来在你的集群中加入更多的节点。

如果集群中某个节点有什么地方出错了（可能是掉线了，不过 Cassandra 无法确定），那么运行 `nodetool` 的时候可以看到一个问号：

```
$ bin/nodetool -h 192.168.1.5 ring
Address       Status   Load       Range                                      Ring
                       112711146095673746066359353163476425700
192.168.1.5   Up       459.26 MB  27647275353297313886547808446514704912    |<--|
192.168.1.7   ?        229.53 MB  112711146095673746066359353163476425700   |-->|
```

6.7.3 多种子节点

Cassandra 允许你指定多个种子节点。种子节点用作其他节点的联络点，这样 Cassandra 可以了解集群的拓扑，也就是说，哪些节点负责哪些键值区间。

默认情况下，配置文件中只有一个 `seed` 项：

```
seeds:
    - 127.0.0.1
```

要在你的环上增加更多的种子节点，只要添加第二个种子元素就行了。我们只需要在配置文件中指出种子节点的 IP 或主机名就可以设置让两个服务器作为种子：

```
seeds:
    - 192.168.1.5
    - 192.168.1.7
```



如果你使用节点自己的 IP 作为种子节点，那么 autobootstrap 元素必须置为 false。一个方法是先启动三个或五个节点作为种子，不使用 autobootstrap，也就是说，手工选择令牌或允许 Cassandra 随机选择令牌。其余的节点将不会再做种子，它们将使用 autobootstrap 加入集群。

接下来，我们需要更新这台机器的监听地址，不能只监听本地还回地址。监听地址是节点互相识别的标识符，并用于所有的内部通信。

```
listen_address: 192.168.1.5
```

最后，我们需要修改另一个地址，RPC 客户端将会在这个地址上进行广播，因为这是其他节点看到本节点的方法。默认情况下，配置文件有一个 `rpc_address` 配置项，使用的是 localhost。我们将会修改这个值为每台机器的真实的 IP 地址：

```
rpc_address: 192.168.1.5
```

这项 `rpc_address` 设置仅用于客户端到 Cassandra 节点的直接连接。



`rpc_adress` 曾经叫做 `ThriftAddress`，不过因为在未来的版本中，可能会使用 Avro 替换 Thrift 作为 RPC 机制，这个参数就改成了一个更为通用的名字。

现在，可以重新启动 Cassandra，并开始启动其他机器上的系统。如果你能成功的在集群中加入多个节点，就会看到类似下面的输出（你的日志可能没有这么多内容，因为我打开了 debug 级的输出）：

```
 INFO 15:45:15,629 Starting up server gossip
 INFO 15:45:15,677 Binding thrift service to /192.168.1.5:9160
 INFO 15:45:15,681 Cassandra starting up...
DEBUG 15:45:16,636 GC for ParNew: 13 ms, 12879912 reclaimed leaving
    11233080 used; max is 1177812992
DEBUG 15:45:16,647 attempting to connect to /192.168.1.7
DEBUG 15:45:17,638 Disseminating load info ...
```

```
 INFO 15:45:19,744 Node /192.168.1.7 is now part of the cluster
DEBUG 15:45:19,746 Node /192.168.1.7 state normal, token
    41654880048427970483049687892424207188
DEBUG 15:45:19,746 No bootstrapping or leaving nodes -> empty pending
    ranges for Keyspace1
 INFO 15:45:20,789 InetAddress /192.168.1.7 is now UP
DEBUG 15:45:20,789 Started hinted handoff for endPoint /192.168.1.7
DEBUG 15:45:20,800 Finished hinted handoff for endpoint /192.168.1.7
DEBUG 15:46:17,638 Disseminating load info ...
```

这些输出显示，我的集群中有两个节点：我刚刚启动的节点正在监听 192.168.1.5，另一个已经启动并且运行的节点，正在监听 192.168.1.7。

6.8 动态加入环

Cassandra 集群中的节点可以随时下线、上线，而不影响到集群的其他部分（假设使用了合理的副本因子和一致性级别）。假设我们启动了 6.7 节提到的一个两节点集群。现在有故障发生，其中一个节点掉线了，我们需要确认集群的其他部分仍然可以工作：

```
INFO 15:34:18,953 Starting up server gossip
INFO 15:34:19,281 Binding thrift service to /192.168.1.7:9160
INFO 15:34:19,343 Cassandra starting up...
INFO 15:45:19,396 Node /192.168.1.5 is now part of the cluster
INFO 15:45:20,176 InetAddress /192.168.1.5 is now UP
INFO 16:13:36,476 error writing to /192.168.1.5
INFO 16:13:40,517 InetAddress /192.168.1.5 is now dead.
INFO 16:21:02,466 error writing to /192.168.1.5
INFO 16:21:02,497 error writing to /192.168.1.5
```

错误消息将一直重复，直到节点 1.5 重新上线位置。运行 `nodetool` 就可以看到，这个节点已经下线，而其他节点仍在工作中。

我们恢复这个节点，并检查 1.7 的日志。显然，Cassandra 已经自动地发现了另一个节点已经回到集群了，并且开始提供服务：

```
INFO 16:33:48,193 error writing to /192.168.1.5
INFO 16:33:51,235 error writing to /192.168.1.5
INFO 16:34:20,126 Standard2 has reached its threshold; switching in a
    fresh Memtable at CommitLogContext(file='/var/lib/cassandra/commitlog\
    CommitLog-127759165782.log', position=752)
INFO 16:34:20,173 Enqueuing flush of Memtable(Standard2)@7481705
INFO 16:34:20,251 Writing Memtable(Standard2)@7481705
INFO 16:34:20,282 LocationInfo has reached its threshold; switching in a
    fresh Memtable at CommitLogContext(file='/var/lib/cassandra/commitlog\
    CommitLog-127759658782.log', position=752)
INFO 16:34:20,298 Enqueuing flush of Memtable(LocationInfo)@24804063
```

```
INFO 16:34:20,579 Completed flushing c:\var\lib\cassandra\data\Keyspace1\
    Standad2-1-Data.db
INFO 16:34:20,594 Writing Memtable(LocationInfo)@24804063
INFO 16:34:20,797 Completed flushing c:\var\lib\cassandra\data\system\
    LocationIfo-2-Data.db
INFO 16:34:58,159 Node /192.168.1.5 has restarted, now UP again
INFO 16:34:58,159 Node /192.168.1.5 state jump to normal
```

这时，节点 1.5 的状态恢复为正常，重新成为了集群的一部分：

```
eben@morpheus$ bin/nodetool -host 192.168.1.5 ring
Address      Status  Load     Range                                    Ring
                     41654880048427970483049687892424207188
192.168.1.5  Up      1.71 KB  20846671262289044293293447172905883342   |<--|
192.168.1.7  Up      2 KB     41654880048427970483049687892424207188   |-->|
```

6.9 安全

默认情况下，Cassandra 允许网络中的任何客户端连接到你的集群上。这并不是说 Cassandra 没有内建的安全机制，而是 Cassandra 默认配置了一个允许所有客户端无需提供任何认证信息就可以登录的鉴权机制。Cassandra 的安全机制是可置换的，这意味着你可以轻易换用其他鉴权机制，或者是写你自己的鉴权机制。

默认插入的鉴权器是 `org.apache.cassandra.auth.AllowAllAuthenticator`。如果要求客户端提供认证信息，可以使用另一个 Cassandra 自带的鉴权器，`org.apache.cassandra.auth.SimpleAuthenticator`。在本节中，我们来看看如何使用这个鉴权器。

6.9.1 使用SimpleAuthenticator

在 config 目录中有两个文件：access.properties 和 passwd.properties。access 文件中存放的键 / 值对用于指定允许访问某个 keyspace 的用户，不同用户间以逗号分隔。示例如下：

```
Keyspace1=jsmith,Elvis Presley,dilbert
```

这里指出了允许访问 `Keyspace1` 的三个用户，用户名中可以有空格。

而 passwd.properties 文件则包含每个用户及其密码。这里是一个 passwd.properties 文件的例子：

```
jsmith=havebadpass
Elvis\ Presley=graceland4evar
dilbert=nomoovertime
```

注意，因为 Elvis Presley 的用户名中包含空格，必须使用一个反斜杠来转义空格。

要使用简单鉴权器，需要在 cassandra.yaml 之中替换 authenticator 元素的内容。从 `org.apache.cassandra.auth.AllowAllAuthenticator` 变为需要登录的实现类的名字：`org.apache.cassandra.auth.SimpleAuthenticator`。

如果还没有正确配置鉴权文件，那么会得到这样一条错误信息：

```
ERROR 10:44:27,928 Fatal error: When using org.apache.cassandra.auth.
    SimpleAuthenticator
access.properties and passwd.properties properties must be defined.
```

这是因为还有一步没有做完：我们必须告诉 Cassandra 入口和 passwords 文件的位置，这要使用 bin/cassandra.in.sh 包含脚本。我们通过在文件尾部加入如下代码，把文件的位置传给 JVM。我的 include 文件现在看起来就像这里的片段，为文件指出完全路径：

```
JVM_OPTS="
        -Dpasswd.properties=/home/eben/books/cassandra/dist/
apache-cassandra-0.7.0-beta1/
conf/passwd.properties
        -Daccess.properties=/home/eben/books/cassandra/dist/
apache-cassandra-0.7.0-beta1/
conf/access.properties"
```

如果你指定了错误的文件位置或名称，服务器会给出相应的出错信息：

```
ERROR 11:13:55,755 Internal error processing login
java.lang.RuntimeException: Authentication table file given by property
passwd.properties
could not be found: /somebadpath/my.properties (No such file or directory)
```

配置成功之后，我们可以使用用户名和密码来登入命令行界面了：

```
[default@unknown] connect localhost/9160
Connected to: "Test Cluster" on localhost/9160
[default@unknown] use Keyspace1 jsmith 'havebadpass'
Authenticated to keyspace: Keyspace1
[jsmith@Keyspace1]
```

如果输入了错误的密码，客户端会提示：

```
[default@unknown] use Keyspace1 jsmith 'havebadpassfdfdfd'
Exception during authentication to the cassandra node: verify keyspace
    exists, and you are using correct credentials.
```

如果输入了不存在的用户名，或试图通过鉴权登入到这个用户没有访问权限的 keyspace，会得到这样的出错信息：

```
[default@unknown] use Keyspace1 dude 'dude'
Login failure. Did you specify 'keyspace', 'username' and 'password'?
[default@unknown]
```

有一点让人不解的是，如果你使用正确的用户名密码，登录一个没有访问权限的keyspace，会看到鉴权信息，但接下来你将不能进行任何操作。如下面的例子，我们进入一个需要鉴权的 keyspace。命令行界面似乎允许鉴权，但我们不能访问任何列：

```
[default@Keyspace1] get Standard1['user123']['name']
Your credentials are not sufficient to perform READONLY operations
[default@Keyspace1]
```

假设已经为这个用户设置了 `'name'` 值，如果提供了正确的认证信息，我们就应该能进行这个操作：

```
[default@Keyspace1] use Keyspace1 eben 'pass'
Authenticated to keyspace: Keyspace1
[eben@Keyspace1] get Standard1['user123']['name']
=> (column=6e616d65, value=bootsy, timestamp=1284316537496000)
[eben@Keyspace1]
```

还能在连接到命令行界面的时候一起输入用户名和密码：

```
eben@morpheus:~/books/cassandra/dist/apache-cassandra-0.7.0-beta1$
    bin/cassandra-cli --host localhost --port 9160
    --username jsmith --password havebadpass --keyspace Keyspace1

Connected to: "Test Cluster" on localhost/9160
Welcome to cassandra CLI.

Type 'help' or '?' for help. Type 'quit' or 'exit' to quit.
[jsmith@Keyspace1] get Standard1['user123']['name']
=> (column=6e616d65, value=bootsy, timestamp=1284316537496000)
[jsmith@Keyspace1]
```

运行查询就可以返回结果。



在 0.6 版本中没有提供通过 Thrift API 的登录操作，所以，如果先启动命令行，再连接到一个需要认证的 keyspace，操作将无法成功。但在 0.7 版本中，已经加入了登录操作，没有这个问题了。



在 0.7 版本中，鉴权器被分为 `IAuthenticator`（负责鉴权）和 `IAuthority`（负责授权）。`SimpleAuthenticator` 类也把授权的功能剥离到 `SimpleAuthority` 类中了。`SimpleAuthenticator` 类只读取 passwd.properties 文件，而 `SimpleAuthority` 只读取 access.properties 文件。

6.9.2 编程鉴权

如果你为 keyspace 设置了鉴权，那么你的客户端应用代码就需要登录操作。可以参考例 6-2 的代码。

例 6-2：编程鉴权登录到一个 keyspace

```
package com.cassandraguide.config;

import java.util.HashMap;
import java.util.Map;

import org.apache.cassandra.thrift.AccessLevel;
import org.apache.cassandra.thrift.AuthenticationRequest;
import org.apache.cassandra.thrift.Cassandra;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

/**
 * 如何连接到设置了 SimpleAuthenticator 的 keyspace
 */
public class AuthExample {

    public static void main(String[] args) throws Exception {

        TTransport tr = new TSocket("localhost", 9160);
        TFramedTransport tf = new TFramedTransport(tr);
        TProtocol proto = new TBinaryProtocol(tf);
        Cassandra.Client client = new Cassandra.Client(proto);
        tr.open();

        AuthenticationRequest authRequest = new AuthenticationRequest();
        Map<String, String> credentials = new HashMap<String, String>();
        credentials.put("username", "jsmith");
        credentials.put("password", "havebadpass");
        authRequest.setCredentials(credentials);

        client.set_keyspace("Keyspace1");

        AccessLevel access = client.login(authRequest);
        System.out.println("ACCESS LEVEL: " + access);
        tr.close();
    }
}
```

在这种情况下，程序会输出 `FULL`，因为这是用户的访问级别。

这里有些问题需要注意。首先，认证信息使用 `username` 和 `password` 作为键，而非以用户名为键、密码为值。其次，需要单独调用 `set_keyspace`，以指定要鉴权的 keyspace。

6.9.3 使用MD5加密

`SimpleAuthenticator` 类有两种密码设置方式：明文文本和 MD5 加密。到目前为止，我们一直使用明文文本方式，也是默认的方式。为了增强安全性，可以考虑 MD5 方式。MD5 是一种加密算法，是“消息摘要算法版本 5（Message-Digest algorithm version 5）”的缩写（版本 5 是说它是版本 4 的一个加强版）。这是一种广泛使用的单向哈希函数，使用输入数据生成一个 128 位的哈希值。值得注意的是，这并非一种非常安全的算法，只是更难被破解而已。

可以通过在 cassandra.in.sh 文件中传入 `passwd.mode` 开关参数来启用 MD5 算法：

```
JVM_OPTS=" \
        -da \
//other stuff...
        -Dpasswd.mode=MD5"
```

现在，如果要使用未加密的密码，就会得到这样一个错误信息：

```
Exception during authentication to the cassandra node, verify you are
using correct credentials.
```

可以使用多种工具，从明文用户名密码通过单向哈希来生成 MD5 加密的版本。这里有一个 Python 程序，可以从明文字符串生成 MD5 哈希：

```
$ python
Python 2.6.5 ...
>>> from hashlib import md5
>>> p = "havebadpass"
>>> h = md5(p).hexdigest()
>>> print h
e1a31eee2136eb73e8e47f9e9d13ab0d
```

现在，在 passwd.properties 文件中使用上面这个加密值来替换 `jsmith` 的密码就可以了。

6.9.4 提供你自己的鉴权算法

如果你有自己的特殊需求，可以为 Cassandra 提供你自己的鉴权算法，比如 Kerberos 认证或加密，或者你希望将密码存在其他位置，比如 LDAP 目录。要创建你自己的鉴权方法，只要实现 `IAuthenticator` 接口即可。这个接口需要提供两个方法，如下：

```
public AccessLevel login(String keyspace, AuthenticationRequest
    authRequest)
    throws AuthenticationException, AuthorizationException;
```

```
public void validateConfiguration() throws ConfigurationException;
```

`login` 方法返回一个 Thrift `AccessLevel` 实例，并指定用户被授权的访问级别。`validateConfiguration` 方法用于检查鉴权机制是否被正确设置了。比如，对于 `SimpleAuthenticator`，就会检查是否指定了 access 和 password 文件。

6.10 杂项设置

还有一些通用设置，不属于之前的那些分类，我把它们都集中在这里了。

- `column_index_size_in_kb`
  这个配置项指定了两次列索引之间，一行可以增长的尺寸，单位为 KB。如果列值都非常大，可能需要提高这个参数的值。超级列没有索引，所以，这个设置只针对列索引。这个设置一般不会设计得很大，因为所有索引数据在每次读访问时都要用到，所以，如果这个值大到超出你的需要，可能会使 Cassandra 的速度受到影响。这对于经常读取部分行的应用场景尤为重要。

- `in_memory_compaction_limit_in_mb`
  这个值代表了在内存中进行压紧操作的行的尺寸限制。如果一行的长度超过了这个值，那么就会分段放在磁盘上进行压紧操作，这会让压紧操作变成更慢的两步操作。这个值默认为 64 MB。在 Cassandra 0.7 之前的版本，这个设置称为 `row_warning_threshold_in_mb`。

- `gc_grace_seconds`
  这个值是对墓碑进行垃圾回收之前等待的秒数。墓碑用于标记已删除数据，这些数据直到到达这个时间阈值才会被永久删除。这个参数的默认设置是 864 000 秒，也就是 10 天。这个时间看起来有些过长了，不过你必须有足够的时间，允许墓碑传播到集群的每个副本上。即使这样，这个时间似乎也太长，因为这个设计的初衷是，你可能会遇到硬件故障，需要时间来让故障节点恢复工作，这个时间让墓碑也可以传播到这些恢复的节点之上。如果宕机节点在墓碑被清除的时候还没有收到墓碑，那么读时修复会导致没有设为墓碑的、未被删除的数据重新被写入回集群中来。

  当然，如果有很多删除操作，可以通过降低这个值来让系统更加整洁，但是这并不会有什么很大的不同。

- `phi_convict_threshold`
  这个值规定了 Cassandra 的故障检测算法的阈值，Phi 值达到这个值之后，Cassandra 会确信这个服务器离线了。这个值的默认设置是 8，通常不需要修改

这个值。但是，如果网络状况不佳，或是有其他原因让你认为 Cassandra 在此有误判，可以通过增加这个值来让节点离线判断更加宽大一些。

> **Phi 阈值与增量故障检测器**
>
> 自从 Cassandra 在 Facebook 开始开发起，就使用了一种称为增量故障检测器（AFD）的节点故障检测机制。这个故障检测器会为每个进程（或节点）给出一个数值。这个值称为 Phi。这个值的输出方式被设计为根据变化的网络状况，自适应地从下向上增长的，而不是一个二元条件，只检查服务器是否正常工作。
>
> 配置文件中的 Phi 判定阈值可以用来调整故障检测器的灵敏度。较低的值提高灵敏度，更高的值则降低灵敏度，这并不是线性的。
>
> Phi 值代表了一个节点可能是下线了的嫌疑程度。Cassandra 这样的使用 AFD 的应用可以为它们输出的 Phi 值指定一个变量条件。Cassandra 使用这个机制，通常可以在 10 秒内发现节点宕机。
>
> 你可以阅读 Phi 增量故障检测的原始论文 http://ddg.jaist.ac.jp/pub/HDY+04.pdf，Cassandra 的设计正是基于此论文的。

6.11 附加工具

本节讨论一些 Cassandra 自带的杂项工具，可以帮助你配置 Cassandra。

6.11.1 查看键值

你可以通过一个名为 `sstablekeys` 的脚本来查看 SSTables 中的键值。要运行这个脚本，只需要使用把要查看键值的 SSTable 的位置输入给脚本即可，如下：

```
eben@morpheus$ bin/sstablekeys /var/lib/cassandra/data/Hotelier/Hotel-1-
    Data.db WARN 10:56:05,928 Schema definitions were defined both locally
    and in cassandra.yaml.
Definitions in cassandra.yaml were ignored.
415a435f303433
415a535f303131
4341535f303231
4e594e5f303432
```

6.11.2 导入之前版本的配置

如果你在 0.6 版本 Cassandra 中使用配置文件定义了一个 keyspace，可能希望将它导入到 0.7 或更高版本中，可以使用一个特殊的 JMX 操作，`StorageService.`

`loadSchemaFromYaml()`。注意，这个方法有两个重要警示：首先这个方法按照设计，只应该使用一次；其次，这个方法可能在 0.8 中不会提供了。

从 0.7 版本开始，用户在 cassandra.yaml 中的 keyspace 定义不会在服务器启动的时候默认导入。所以，你在服务器第一次启动的时候会看到这条消息：

```
INFO config.DatabaseDescriptor: Found table data in data directories.
Consider using JMX to call org.apache.cassandra.service.StorageService.
loadSchemaFromYaml().
```

为了导入数据，你需要调用 `loadSchemaFromYaml` JMX 操作。不过在调用这个命令之前，还需要一些准备工作。首先，要从 Sourceforge.net 下载 mx4j-tools.jar。下载 ZIP 文件并解压到任意目录。之后把 lib 目录中的 JAR 重命名为 mx4j-tools.jar，放到 Cassandra 的 lib 目录。这样你就可以进行 JMX 连接，来与 Cassandra 进行多种 JMX 交互了。我们会在后面讨论如何使用 JMX 与 Cassandra 交互，不过这里只希望装入 keyspace 进行测试。现在，在加入 JAR 包后重启 Cassandra。

接下来，打开一个终端，输入命令 jconsole。这会启动一个 GUI，打开 Java 自带的 JConsole 工具。这个工具可以内省 Java 虚拟机，并允许你查看运行时的数据、调用通过 JMX 暴露出来的操作。

要加载 cassandra.yaml 定义的 keyspace，单击 JConsole 图形界面的 MBeans 标签。展开 `org.apache.cassandra.service` bean，接着点开 `StorageService`，然后是 Operations。点开 `loadSchemaFromYAML` 操作并点操作的按钮。这将在 Cassandra 上执行命令，加载存放在 cassandra.yaml 文件中的 schema。

要进行这个将 0.6 的配置导入到 0.7 的一次性操作，需要使用 `StorageService` MBean 的 `loadSchemaFromYaml` 操作。操作方法如图 6-4 所示。

现在，你应该可以在日志中看到类似这样的信息：

```
17:35:47 INFO thrift.CassandraDaemon: Cassandra starting up...
17:35:48 INFO utils.Mx4jTool: mx4j successfuly loaded
    HttpAdaptor version 3.0.2 started on port 8081
17:40:43 INFO config.DatabaseDescriptor: UTF8Type
17:40:43 INFO config.DatabaseDescriptor: BytesType
17:40:43 INFO config.DatabaseDescriptor: UTF8Type
17:40:43 INFO config.DatabaseDescriptor: TimeUUIDType
17:40:43 INFO config.DatabaseDescriptor: BytesType
17:40:43 INFO config.DatabaseDescriptor: BytesType
17:40:43 INFO config.DatabaseDescriptor:
```

DatabaseDescriptor 日志非常有用，它显示出了加载 schema 的操作。

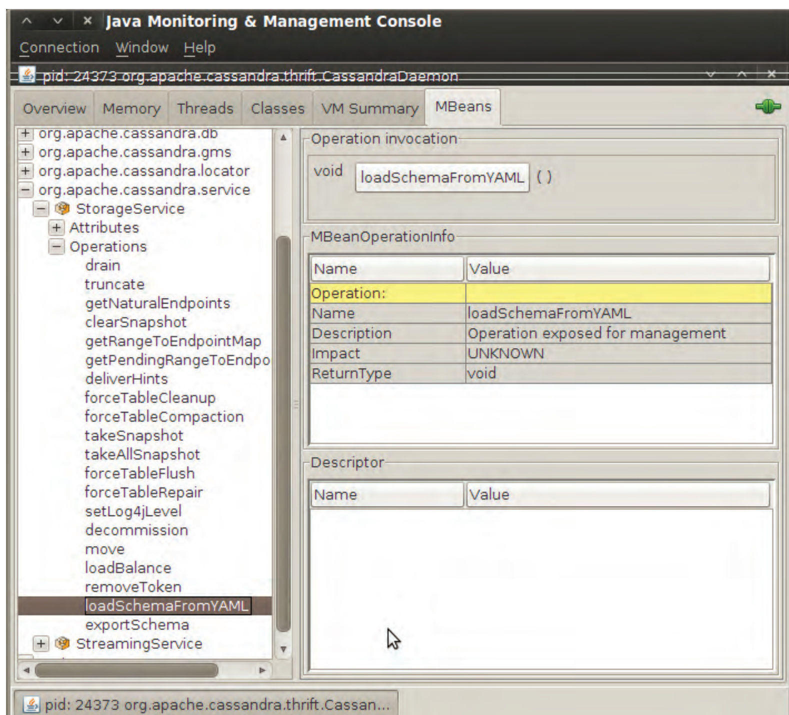


图 6-4：使用 jconsole 工具加载 cassandra.yaml 中定义的老的 keyspace

你还可以使用 `org.apache.cassandra.config.Converter` 类来帮助你将 storage-conf.xml 配置文件转化为 cassandra.yaml 文件。如果你从 0.6 升级到 0.7，这应该是升级的第一步。要运行这个转化器，得使用 bin 目录中的 config-converter 脚本。它会读入老配置文件，转化并将内容导出。



如果需要导入导出数据，Cassandra 有两个使用 JSON 的脚本：一个用于导入 JSON 到 SSTable，另一个用于将 SSTable 导出到 JSON。

6.12 小结

本章中，我们看了如何设置 Cassandra，包括使用 API 的新的动态配置方法，以及一些挖掘原始数据文件的自带工具。我们还看到了如何设置副本因子和 Snitch，以及如何使用合适的副本放置策略。

如果你更喜欢使用图形界面而非命令行界面，可以了解使用 Chiton 设置 Cassandra，这是一个 Brandon Williams 用 Python 写的基于 GTK 的 Cassandra 浏览器。这个项目可以在 http://github.com/driftx/chiton 找到。

第 7 章

读写数据

现在，我们已经了解了 Cassandra 的数据模型，将继续研究 Cassandra 读写数据时进行的各种查询操作，本章中将使用 Cassandra 0.6.7-beta1，这是本章写作时的最新版本。

7.1 Cassandra与RDBMS查询的不同

Cassandra 的模型的查询方法与 RDBMS 的有很多不同之处，这些都非常重要，需要时时牢记。

7.1.1 没有Update查询

Cassandra 之中，并没有“update”（更新）这个初指令概念，也就是说，没有一个称为“update”的客户端查询命令。不过，你可以通过对一个已经存在的行键值执行一次插入操作来完成同样的工作。如果对一个已经存在的键值执行一个插入语句，Cassandra 会覆盖已经存在的匹配的列，而如果查询中包含了这个行键值下还不存在的列，那么新的列会被插入。这个操作是完全无缝的。

7.1.2 记录级的写原子性

Cassandra 在每次写操作时，自动提供记录级的原子性。而在 RDBMS 中，你必须指定使用行级锁。虽然 Cassandra 提供列族级的原子性，但并不保证隔离性。

7.1.3 不支持服务端事务

因为你得将表反范式化来创建第二索引，所以可能需要将数据插入到两个或更多个

表中（一份放入主表，其他的用作反向索引或第二索引）。这意味着需要执行两个插入语句。所以，如果应用场景是如此这般，要是一次插入操作失败了，就不得不删除已经插入的数据，来进行手工的“回滚”了。

更准确地说，对于 Cassandra 不支持事务这点，或许只是因为这本身就不是 Cassandra 的设计目标。

7.1.4 没有重复键值

在 SQL 数据库中，如果没有定义某列作为唯一的主键，可以插入多行一样的值。但在 Cassandra 中，这是不可能的。如果对列族中一个已经存在的键值写入新值，原来已经存在的列就将被覆盖，而先前不存在的列也将被加入其中。

7.2 写操作的基本属性

Cassandra 的写操作有一些值得注意的基本属性。首先，Cassandra 的写操作非常快，这是因为它的设计允许在写时不进行任何磁盘读或定位（seek）操作。memtable 和 SSTable 让 Cassandra 可以不必在写时进行这些操作，正是这些操作让很多数据库变慢的。Cassandra 中的所有写操作都是追加写的。

因为数据库的 commit log 和提示移交设计，Cassandra 总是可写的，而且，在一个列族内，写操作总是最小单元的。

7.3 一致性级别

Cassandra 的可调一致性级别意味着可以在查询中指定需要多少一致性。高一致性级别意味着更多的节点需要响应这次查询操作，这样可以提供更高的一致性保障，确保每个副本都是同样的值。如果两个节点的响应有不同的时间戳，最新的值将会胜出，并返回给客户端。在后台，Cassandra 接下来还会进行读时修复操作：它知道有些副本存在过期的数据，并会使用最新的数据来更新这些副本，保证它们是一致的。

Cassandra 中，可以在多个一致性级别中进行选择，相对于写操作，一致性级别对于读操作的处理上差异更大。表 7-1 列出了可用的一致性级别，以及这些不同的一致性级别对于读查询的意义。



一致性级别是基于配置文件中指定的副本因子的，而不是系统中的节点总数。

表7-1　读一致性级别

一致性级别	含　义
ZERO	不支持。你不能在读操作时指定 CL.ZERO，因为这没有意义。这等于说“不要从任何节点给我数据”
ANY	不支持。应使用 CL.ONE
ONE	当第一个节点响应查询时，立刻返回该响应的值。同时会创建一个后台线程，检查这个记录的其他副本。如果哪个副本已经过期了，接下来就会进行读时修复，以确保所有副本都拥有最新的值
QUORUM	查询所有节点。当大部分副本（(副本因子 /2）+1）返回的时候，把时间戳最新的值返回客户端。之后，如果必要则在后台对其余副本进行读时修复
ALL	查询所有节点。等待所有节点响应，并把时间戳最新的记录返回给客户端。之后，如果必要的话，在后台进行一次读时修复。如果有任何节点没有响应，读操作都会失败

正如你在表中看到的，一致性级别 ZERO 和 ANY 对于读操作是不支持的。注意，一致性级别 ONE 意味着客户端将得到第一个给出响应的节点的值——即使这个值是过期的。因为在记录返回给客户端之后会进行读时修复操作，所以，接下来的读操作都可以得到一致的值，不论响应的节点是哪一个。

另一个值得注意的是 ALL。如果指定了 CL.ALL，就是在要求所有副本必须都给出响应，如果任何持有这个记录的节点宕机，或由于其他任何原因发生超时，这个读操作都将失败。



如果一个节点在一个指定的时间内未能响应查询，则被判断为无响应。这个时间由配置文件中的 rpc_timeout_in_ms 设定，默认值为 10 秒。

你同样可以为写操作指定一致性级别，不过它们的含义有很大区别。不同一致性级别对于写操作的含义如表 7-2 所示。

表7-2　写一致性级别

一致性级别	含　义
ZERO	在写数据被记录之前就返回；写操作将会在一个后台线程中异步完成，无法确保写操作一定成功
ANY	保证数据至少已经写到一个节点上了，提示也被看做是一个成功的写入
ONE	保证在返回时，数据至少已经写入到一个节点的commit log和memtable之中了
QUORUM	保证多数副本（(副本因子/2）+1）已经接收到数据了
ALL	保证在返回时副本因子指定数量的节点都接收到数据了。如果某个副本对写操作无响应，则写操作会失败

最值得一提的写一致性级别就是 ANY。这个级别意味着确保写操作至少到达了一个节点，但允许提示被看做是一次成功写入。也就是说，如果你在进行写操作，而目标节点刚好都宕机了，那么服务器将会自己记录下来，称为提示（hint），一直保持

到目标节点恢复为止。当服务器发现一个节点恢复后，会检查是否存有该节点的提示，如果有，就会将值写入到恢复的节点上。在很多情况下，生成提示的节点并不是存储提示的节点，而是会把提示发送给宕机节点的一个没有此副本的邻居节点。

写操作时使用一致性级别 `ONE`，意味着写操作将被写入到 commit log 和 memtable。也就是说，`CL.ONE` 写入的数据是持久化的，要达到高性能、持久化的写入，这是可用的最低的一致性级别。如果这个节点在写操作之后立刻掉线，那么数据仍将存在在 commit log 之中，当节点重新恢复后可以重放操作，保证数据依然存在。

不论是读操作还是写操作，`ZERO`、`ANY` 和 `ONE` 都被划分为弱一致性，而 `QUORUM` 和 `ALL` 则属于强一致性。因为 Cassandra 中的读写操作可以分别指定一致性级别，所以 Cassandra 的一致性被认为是可调的。在 Cassandra 之中，要达到强一致性，可以使用如下这个流行的公式：R+W>N= 强一致性。在公式中，R、W、N 分别是读副本数、写副本数和副本因子，这时，所有客户端都可以得到最新的写入结果，即可以得到强一致性。

7.4 读操作的基本属性

Cassandra 的读数据操作也有一些值得一提的基本属性。首先，读数据非常容易，因为你可以连接集群中的任意一个节点来进行读操作，不需要知道哪个特定的节点负责具体所需的数据的副本。如果客户端连接的节点没有所需的数据，它会自己扮演协调节点的角色，根据令牌范围，从拥有数据的节点读取数据。

要完成读操作，Cassandra 需要进行磁盘定位操作，但可以通过增加内存来加速读操作。如果你发现操作系统在读操作时有很多分页操作，那么增加内存可能可以提高性能（通常来说，打开 Cassandra 的各种缓存效果会更好）。Cassandra 必须要等待一些同步响应（根据一致性级别和副本因子的要求），之后还要根据需要进行读时修复。

基于上述这些原因，读操作肯定会比写操作更慢。分区器并不影响读操作的速度。在进行区域查询时，使用 OPP 会显著变快，因为它让你更容易判断哪个节点没有所需的数据。分区器的职责是进行一致哈希，将键值映射到节点。此外，你还可以选择行缓存和键值缓存策略来提升性能（参考第 11 章）。

7.5 API

本节会给出 Cassandra API 的一个基本概述，这样我们开始读写数据的时候，这些生僻的名词不会看起来那么难懂。我们已经知道了列、超级列和列族这些概念了：

列族包含了列或是超级列；超级列只包含列（超级列里不能再嵌套定义超级列了）；列包含名 / 值对和时间戳。

在关系型数据库中，SELECT（选取）、INSERT（插入）、UPDATE（更新）和 DELETE（删除）正如它们本身通俗的含义一样。但 Cassandra API 中的结构却没有这么直接，让新手可能有些费解，毕竟现实生活中并没有一个“切片区间（slice range)”，对这些名词需要一个适应的过程。

首先你需要理解两个基本概念：区间（range）和切片（slice）。许多查询都使用这两个名词定义，而它们起先可能有些让人迷惑。



列是按照类型排序的（由 `CompareWith` 指定），而行是按照它们的分区器排序的。

区间和切片

区间基本源于数学的“区间”概念，当你有一组有序元素时，可以通过指定开始元素和结束元素来指定一个子集。区间指从开始到结束之间的所有元素，无一例外。

区间通常指（行）键值的范围。而切片则用来指一行之中的一个范围内的列。

区间依据列族的比较器（comparator）来工作。也就是说，给定列 `a`、`b`、`c`、`d` 和 `e`，其上的区间 `(a,c)` 包含列 `a`、`b` 和 `c`。所以，如果你有 1000 个名字为长整型的列，可以指定查询名字是 35 到 45 这个区间之内的列。通过使用区间，可以取出所有指定的范围（称为一个*区间切片*）之内的列，或者也可以同样批量更新一个区间之内的元素。

一行中可能会拥有上百个列，但你不太可能希望用一个查询把它们全都取出来。列是有序存储的，于是提供了区间查询这一手段，可以一次取出名称在一定范围之内的列。



区间查询需要使用有序分区器（`OrderPreservingPartitioner`），这样键值就可以有序返回，其顺序是由分区器使用的码页（UTF-8 或 en_US）来定义的。

当使用随机分区器并指定区间查询时，无法指定比“all”更窄的区间。这显然会有很大开销，因为可能会包含更多的网络传输。而且可能导致错过键值。因为同时可能正在进行一次更新，这时行扫描将会错失在你当前处理的内容之前放到索引中的更新。

还有一点可能会引起误解。当你使用随机分区器的时候，必须明白，区间查询会首先对键值求哈希值。所以，如果使用从“Alice”到“Alison”之间的范围，查询将对每个键值求哈希值，返回值并不是 Alice 到 Alison 之间的自然值，而是这两个值的哈希值之间的键值范围。

如下是使用随机分区器时，查找某特定键值的读操作的基本流程。首先，对键值求哈希值，随后客户端将会连上集群中的一个任意节点。这个节点将会把请求路由到拥有这个键值的节点。首先会查找 memtable，查看这个键值是否存在。如果没有发现，会从最新的文件开始，通过 Bloom filter 查询每个文件。一旦通过 Bloom filter 找到键值，就会打开相应的数据文件来查找列值。

7.6 设置与插入数据

我们首先来看插入，因为你需要数据库里有些内容来进行查询。在本节中，我们建立一个 Java 项目，并学习一个完整的例子，来看如何插入数据，再将数据读出。

首先，从 http://cassandra.apache.org 下载 Cassandra。二进制版本很适合于起步应用，如果你遇到问题的话，可以参考第 2 章。

现在，让我们创建一个新项目来测试一下我们的成绩吧。这里使用 Eclipse 中的 Java 项目。首先，创建新项目，然后将一些 JAR 包添加到 classpath 之中：Log4J 的 JAR 包用于输出日志（别忘了为你自己的类设置相应的属性文件）；Thrift 库名为 libthrift-r917130.jar，其中包含了 `org.apache.thrift` 类；Cassandra 的 JAR 包 apache-cassandra-x.x.x.jar，其中包含了 `org.apache.cassandra` 包下的各个类。我们还需要 SLF4J 日志工具（API 和实现的 JAR 包都需要），这是 Cassandra 需要的。最后，添加 JVM 的参数，来把 log4j.properties 文件加入到 classpath 之中：

```
-Dlog4j.configuration=file:///home/eben/books/cassandra/log4j.properties
```



在 Eclipse 中，可以通过创建一个新的运行配置（Run Configuration）来添加 log4j.properties 文件。单击参数（Arguments）标签，然后在虚拟机参数（VM Argument）文本域，添加上面的代码样例里指定的参数——当然，你需要改成你自己的属性文件的路径。

我的 log4j.properties 文件如下所示：

```
# output messages into a rolling log file as well as stdout
log4j.rootLogger=DEBUG,stdout,R

# stdout
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p %d{HH:mm:ss,SSS} %m%n

# rolling log file
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.file.maxFileSize=5MB
log4j.appender.file.maxBackupIndex=5
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%5p [%t] %d{ISO8601} %C %F (line
    %L) %m%n

# This points to your logs directory
log4j.appender.R.File=cass-client.log
```

为了简单起见，我们使用了默认 keyspace 和配置。现在启动服务器，并创建一个如例 7-1 所示的类。这个例子将要建立一个到 Cassandra 的连接，并写入一个新行，包含 `name` 和 `age` 两个列。我们之后会读取这行的一个列值，再之后读取整行。

例 7-1：SimpleWriteRead.java

```
package com.cassandraguide.rw;

import java.io.UnsupportedEncodingException;
import java.util.List;

import org.apache.cassandra.thrift.Cassandra;
import org.apache.cassandra.thrift.Clock;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ColumnPath;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.cassandra.thrift.InvalidRequestException;
import org.apache.cassandra.thrift.NotFoundException;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.cassandra.thrift.TimedOutException;
import org.apache.cassandra.thrift.UnavailableException;
import org.apache.log4j.Logger;
import org.apache.thrift.TException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

public class SimpleWriteRead {

    private static final Logger LOG = Logger.getLogger(SimpleWriteRead.class);

    //设置一些常量
    private static final String UTF8 = "UTF8";
    private static final String HOST = "localhost";
```

```
private static final int PORT = 9160;
private static final ConsistencyLevel CL = ConsistencyLevel.ONE;

//不处理这之中的异常
public static void main(String[] args) throws UnsupportedEncoding-
    Exception, InvalidRequestException, UnavailableException,
        TimedOutException, TException, NotFoundException {

    TTransport tr = new TSocket(HOST, PORT);
    //0.7之中 framed transport 是默认设置
    TFramedTransport tf = new TFramedTransport(tr);
    TProtocol proto = new TBinaryProtocol(tf);
    Cassandra.Client client = new Cassandra.Client(proto);
    tf.open();
    client.set_keyspace("Keyspace1");

    String cfName = "Standard1";
    byte[] userIDKey = "1".getBytes(); //这是行键值

    Clock clock = new Clock(System.currentTimeMillis());

    //创建名称列的一个表述
    ColumnPath colPathName = new ColumnPath(cfName);
    colPathName.setColumn("name".getBytes(UTF8));

    ColumnParent cp = new ColumnParent(cfName);

    //插入姓名列
    LOG.debug("Inserting row for key " + new String(userIDKey));
    client.insert(userIDKey, cp,
            new Column("name".getBytes(UTF8),
                    "George Clinton".getBytes(), clock), CL);

    //插入年龄列
    client.insert(userIDKey, cp,
            new Column("age".getBytes(UTF8),
                    "69".getBytes(), clock), CL);

    LOG.debug("Row insert done.");

    // 只读姓名列
    LOG.debug("Reading Name Column:");
    Column col = client.get(userIDKey, colPathName,
        CL).getColumn();

    LOG.debug("Column name: " + new String(col.name, UTF8));
    LOG.debug("Column value: " + new String(col.value, UTF8));
    LOG.debug("Column timestamp: " + col.clock.timestamp);

    //创建一个片，来表示要读取的列范围的起始值和终止值
    //这里使用的是all
    SlicePredicate predicate = new SlicePredicate();
    SliceRange sliceRange = new SliceRange();
```

```
        sliceRange.setStart(new byte[0]);
        sliceRange.setFinish(new byte[0]);
        predicate.setSlice_range(sliceRange);

        LOG.debug("Complete Row:");
        // 读取这行的所有列
        ColumnParent parent = new ColumnParent(cfName);
        List<ColumnOrSuperColumn> results =
            client.get_slice(userIDKey,
                    parent, predicate, CL);


        //通过循环读取各列，输出各列的值
        for (ColumnOrSuperColumn result : results) {
            Column column = result.column;
            LOG.debug(new String(column.name, UTF8) + " : "
                    + new String(column.value, UTF8));
        }
        tf.close();

        LOG.debug("All done.");
    }
}
```

这个例子的输出结果如下所示：

```
DEBUG 14:02:09,572 Inserting row for key 1
DEBUG 14:02:09,580 Row insert done.
DEBUG 14:02:09,580 Reading Name Column:
DEBUG 14:02:09,585 Column name: name
DEBUG 14:02:09,586 Column value: George Clinton
DEBUG 14:02:09,586 Column timestamp: 1284325329569
DEBUG 14:02:09,589 Complete Row:
DEBUG 14:02:09,594 age : 69
DEBUG 14:02:09,594 name : George Clinton
DEBUG 14:02:09,594 All done.
```



补充一点，这不是 Cassandra 所特有的特性，使用 Java，你都可以很方便地利用 `Date` 对象封装 `long` 型的时间戳，得到更加用户友好的表达形式，用法如下：`new Date(col.timestamp);`。

现在，我们展开介绍所做的工作。首先，我们连接到 Cassandra 服务器：

```
TTransport tr = new TSocket(HOST, PORT);
//0.7 中新的默认方法是分帧传输
TFramedTransport tf = new TFramedTransport(tr);
TProtocol proto = new TBinaryProtocol(tf);
Cassandra.Client client = new Cassandra.Client(proto);
tf.open();
client.set_keyspace("Keyspace1");
```

这里我们使用了分帧传输（framed transport），这是 Cassandra 0.7 中的默认设置。上述代码连接到了 Cassandra 的一个名为 `Keyspace1` 的 keyspace。

然后，我们创建列族的名称、用做行键值的值，以及插入时需要指定的时钟：

```
String cfName = "Standard1";
byte[] userIDKey = "1".getBytes(); // 行键值

Clock clock = new Clock(System.currentTimeMillis());
```

接下来，我们使用带有列路径的客户端对象，插入新值：

```
ColumnParent cp = new ColumnParent(cfName);

// 插入 name 列
LOG.debug("Inserting row for key " + new String(userIDKey));
client.insert(userIDKey, cp,
        new Column("name".getBytes(UTF8),
                "George Clinton".getBytes(), clock), CL);
```

插入操作需要行键值和列对象，列对象中包含列名和我们期望赋给它的值。同时我们还需要指定一个时钟值，用以标记插入执行的时间，以及指定的一致性级别。

之后，重复一次类似的工作，在同一行插入 age 列，值为 69：

```
client.insert(userIDKey, cp,
        new Column("age".getBytes(UTF8),
                "69".getBytes(), clock), CL);
```

这里，我们已经在一行里插入了两列，并且已经准备好读出来进行检验了。

要验证插入的值是否被正确读出，可使用客户端的 `get` 方法，传入要读取的行键值和列路径（如下代码是 `name` 列），并指定这个操作需要的一致性级别：

```
ColumnPath colPathName = new ColumnPath(cfName);
colPathName.setColumn("name".getBytes(UTF8));
Column col = client.get(userIDKey, colPathName,
        CL).getColumn();

LOG.debug("Column name: " + new String(col.name, UTF8));
LOG.debug("Column value: " + new String(col.value, UTF8));
LOG.debug("Column timestamp: " + col.clock.timestamp);
```

每列的值都有其自己的时间戳（包装为一个时钟对象），因为列（而非行）是一个原子级的单元。如果你使用关系型数据库的时候，习惯用时间戳列来记录最近被更新的时间，对这一点可能感到有些不适应。在 Cassandra 中，没有标记行被最后更新时间的东西，更新总是以列为粒度的。

因为 Cassandra 会为列名和值返回一个字节数组，我们使用这个字节数组创建一个字符串对象，这样就可以在应用中使用它了（比如这里的写日志操作）。时钟对象存储为一个长整型（代表从 Unix 纪元开始时计算的毫秒数），如果需要，我们可以使用 `java.util.Date` 对象来包装它。

使用 get 方法，并指定列路径和其他参数，可以读出一个列的值。而现在，我们将取出一行之中的一个列“区间”（称为*切片*）。这里，我们使用切片谓词：

```
SlicePredicate predicate = new SlicePredicate();
SliceRange sliceRange = new SliceRange();
sliceRange.setStart(new byte[0]);
sliceRange.setFinish(new byte[0]);
predicate.setSlice_range(sliceRange);
```

切片谓词（slice predicate）是一个容器对象，使用这个对象，可以通过指定起始点和终止点来指定我们想读的列的范围。这里，指定开始点和终止点都是 new byte[0]，表示希望读取所有列。

现在，谓词已经设置好了，可以运行这个区间切片查询，读取一行中的所有列，然后就可以用循环来逐个处理了：

```
ColumnParent parent = new ColumnParent(cfName);
List<ColumnOrSuperColumn> results =
  client.get_slice(userIDKey, parent, predicate, CL);
```

如上，这个 get_slice 查询使用了我们创建的谓词，同时还有两个新参数：ColumnOrSuperColumn 类和一个 column parent。ColumnOrSuperColumn 类正如它的名字所指的：代表了 Thrift 返回的一个列或一个超级列。Thrift 不支持继承，所以这个类用于将类和超级类打包在一个对象中（具体是什么取决于查询）。如果返回的是一个列，客户端就只读取其值；而如果返回的是超级列，客户端就会从超级列中再读取一列。

column parent 是到一组列的上层（列族或超级列）的路径。因为我们要通过 get_slice 接收一组列，需要指定容纳查找的这组列的列族。现在，可以针对这行来循环处理这些列了，打印输出每列的三个属性，之后关闭连接：

```
for (ColumnOrSuperColumn result : results) {
  Column column = result.column;

  LOG.debug(new String(column.name, UTF8) + " : "
    + new String(column.value, UTF8));
}
tf.close();
```

你可以使用 insert 操作来添加值或是覆盖已有值。要更新一个值，只要对同样的键值使用新的列值，执行一次 insert 就可以了。

你还可以一次插入多个值，这将在 7.13 节中介绍。

7.7 使用简单的get

使用 get 操作可以依据列路径取出列或超级列：

```
ColumnOrSuperColumn get(byte[] key, ColumnPath column_path,
  ConsistencyLevel consistency_level)
```

例 7-2 演示了如何操作。

例 7-2：使用 get 操作

```
package com.cassandraguide.rw;

//这里省略了 imports

public class GetExample {

    private static final Logger LOG = Logger.getLogger(GetExample.class);

    private static final String UTF8 = "UTF8";
    private static final String HOST = "localhost";
    private static final int PORT = 9160;
    private static final ConsistencyLevel CL = ConsistencyLevel.ONE;

    public static void main(String[] args) throws UnsupportedEncodingException,
            InvalidRequestException, UnavailableException, TimedOutException,
            TException, NotFoundException {

        TTransport tr = new TSocket(HOST, PORT);
        //0.7 中新的默认方法是分帧传输
        TFramedTransport tf = new TFramedTransport(tr);
        TProtocol proto = new TBinaryProtocol(tf);

        Cassandra.Client client = new Cassandra.Client(proto);
        tf.open();
        client.set_keyspace("Keyspace1");

        String cfName = "Standard1";
        byte[] userIDKey = "1".getBytes(); //行键值

        Clock clock = new Clock(System.currentTimeMillis());

        //创建 name 列的描述
        ColumnParent cp = new ColumnParent(cfName);

        //插入 name 列
        LOG.debug("Inserting row for key " + new String(userIDKey));
        client.insert(userIDKey, cp,
                new Column("name".getBytes(UTF8),
                        "George Clinton".getBytes(), clock), CL);

        LOG.debug("Row insert done.");

        /** 进行 get 操作 */

        LOG.debug("Get result:");
        // 读取这行的所有列
        ColumnPath path = new ColumnPath();
        path.column_family = cfName;
```

```
        path.column = "name".getBytes();

        ColumnOrSuperColumn cosc = client.get(userIDKey, path, CL);
        Column column = cosc.column;
        LOG.debug(new String(column.name, UTF8) + " : "
                + new String(column.value, UTF8));
        // get 完成

        tr.close();

        LOG.debug("All done.");
    }
}
```

这里，我们先进行了一次插入，以便可以取出数据。我们创建一个 client 对象，并调用了这个对象的 get 方法，这个方法有三个参数：行键值、列路径、一致性级别。其中，列路径设置了要查找的列名称。注意，列名称和值都是二进制的（对于 Java 客户端来说，是字节数组），所以我们在查询时必须将字符串列名转化成字节数组。之后得到的列的值同样也是字节数组，为了处理结果，我们还要再将它转化成字符串。

在本例中，我们插入了 name 和 age 两个列，但因为列路径只指定了一个想查询的列，所以我们只得到了 age。输出结果如下所示：

```
DEBUG 14:36:42,265 Inserting row for key 1
DEBUG 14:36:42,273 Row insert done.
DEBUG 14:36:42,273 Get result:
DEBUG 14:36:42,282 name : George Clinton
DEBUG 14:36:42,282 All done.
```

7.8 数据准备

这里，我们使用命令行接口来创建一些有不同列的键值，以便后面查询：

```
[default@Keyspace1] set Standard1['k1']['a']='1'
Value inserted.
[default@Keyspace1] set Standard1['k1']['b']='2'
Value inserted.
[default@Keyspace1] set Standard1['k1']['c']='3'
Value inserted.
[default@Keyspace1] set Standard1['k2']['a']='2.1'
Value inserted.
[default@Keyspace1] set Standard1['k2']['b']='2.2'
```

现在，我们有两行数据，第一行有三列，第二行有两列。

7.9 切片谓词

切片谓词（slice predicate）可以用于读和写操作，是用于指定一组列的限定词。你

可以使用两种方式来指定切片谓词：一组列名或是一个切片区间。如果你知道所要取出的几个列的名称，那么可以明确地指定名称；而如果不知道它们的确切名称，或者由于其他需要得取出一个范围之内的所有列，那么就可以使用切片区间来指定这个范围。



简单起见，我使用了包含的例子。但是，一行之中有很多很多列对 Cassandra 来说是家常便饭，所以别被这些例子误导。Cassandra 可以存储海量的数据，在 Cassandra 0.7 之中，每行可以容纳 20 亿列之多。

要使用切片谓词，首先要创建包含你想获取的列名的谓词对象，然后将它传给读操作。

7.9.1 使用get_slice读取特定列名

如果你希望取出一行中名叫“a”和“b”的列，可以使用指定列名的谓词。

使用 `get_slice` 操作，可以取出列族或是超级列中的一组列。它会根据列名或列名的区间取出值来，返回列或者超级列。`get_slice` 操作的声明形式如下：

```
List<ColumnOrSuperColumn> get_slice(byte[] key,
ColumnParent column_parent, SlicePredicate predicate, ConsistencyLevel cl)
```

使用方法如例 7-3 所示。

例 7-3：SlicePredicate.java

```
package com.cassandraguide.rw;

// imports 已省略

public class SlicePredicateExample {

    public static void main(String[] args) throws Exception {
        Connector conn = new Connector();
        Cassandra.Client client = conn.connect();

        SlicePredicate predicate = new SlicePredicate();
        List<byte[]> colNames = new ArrayList<byte[]>();
        colNames.add("a".getBytes());
        colNames.add("b".getBytes());
        predicate.column_names = colNames;

        ColumnParent parent = new ColumnParent("Standard1");

        byte[] key = "k1".getBytes();
        List<ColumnOrSuperColumn> results =
            client.get_slice(key, parent, predicate, ConsistencyLevel.ONE);
```

```
        for (ColumnOrSuperColumn cosc : results) {
            Column c = cosc.column;
            System.out.println(new String(c.name, "UTF-8") + " : "
                    + new String(c.value, "UTF-8"));
        }

        conn.close();

        System.out.println("All done.");
    }
}
```

在这个例子中，只有指定的列会被取出，其他列则被忽略。这个查询返回一个 `ColumnOrSuperColumn` 对象的列表。因为我们知道所查询的是普通的列族，所以直接从底层 RPC 机制（Thrift）返回的 `ColumnOrSuperColumn` 数据结构中取出列的值，最后在一个循环中读取这些列的名称和值。

例子的输出如下所示：

```
a : 1
b : 2
All done.
```

7.9.2 通过切片区间获取一组列

有时你不希望指定所要取出的每个列，一个可能的原因是要取出很多列，也有可能是因为你不知道所有这些列的名称。

要读取一行中一个指定区间的列，可以指定开始和结束的列，Cassandra 就会给你整个区间内的列，包括两端的列，区间范围的确定是根据列族的比较器排序而得到的。首先创建切片谓词，之后创建区间，再把谓词传递给读操作之前设置区间。这是一个例子：

```
SlicePredicate predicate = new SlicePredicate();
SliceRange sliceRange = new SliceRange();
sliceRange.setStart("age".getBytes());
sliceRange.setFinish("name".getBytes());
predicate.setSlice_range(sliceRange);
```

在执行 `get_slice` 操作时，查询会返回这两个指定的列，以及所有通过比较器排序比较落在两个列之间的列（可能依照文字、数字或其他什么东西来排序）。比如，如果这行也有一个“email”列，它也会被返回。

你必须根据比较器来给出列名，注意开始列和结束列的顺序。比如，如果以 `name` 列开始、以 `age` 列结束，就会抛出一个异常：

```
InvalidRequestException(why:range finish must come after start in the order
  of traversal)
```



别忘记了，“返回一列”并非意味着像 SQL 一样得到列的值，而是得到完整的列数据结构，包括名称、值和时间戳。

计数

可以使用 `Slice Range` 结构的计数（count）属性来限制切片区间返回行的数量。假设我们有一个几百列的行，可以指定一个包含很多列的区间，但限制只返回结果的前 10 列，如下：

```
SliceRange sliceRange = new SliceRange();
sliceRange.setStart("a".getBytes());
sliceRange.setFinish("d".getBytes());
sliceRange.count = 10;
```

再强调一次，“第一”列要依照列族的比较器指定的顺序而定。

逆序

你还可以通过设置切片区间的 `reversed=true` 属性来取出呈现逆序的列。如果有 `age`、`email` 和 `name` 三个列，那么设置 `reversed` 为 `true`，得到的列的顺序就是 `name`、`email` 和 `age`。

7.9.3 取出一行中的所有列

要读出一行中的所有列，也需要使用有切片区间的谓词，但只要给区间的起始和结束参数一个空字节数组就可以了，像这样：

```
SlicePredicate predicate = new SlicePredicate();
SliceRange sliceRange = new SliceRange();
sliceRange.setStart(new byte[0]);
sliceRange.setFinish(new byte[0]);
predicate.setSlice_range(sliceRange);
```

然后，就可以把这个增加的谓词对象和其他必要参数（如一致性级别之类的）一起传给 `get_slice` 操作了。

7.10 get_range_slices

类似于使用区间来访问一组列，我们也可以访问一个键值或令牌区间。可以把一个 `KeyRange` 对象传给一个 `get_range_slices` 操作，来定义一组要访问的键值。

这里的一个不同是，作为 `get_range_slices` 的参数，`KeyRange` 数据结构可以指定键值，也可以指定令牌。键值区间包含开始点，而令牌不包括开始点。因为令牌是环状分布的，所以结束令牌可以小于起始令牌。

API 中定义了 `get_range_slices` 操作，其操作方式大致如下：

```
List<KeySlice> results = client.get_range_slices(parent, predicate, keyRange,
    ConsistencyLevel);
```

例 7-4 给出了一个完整的使用区间切片的例子。

例 7-4：GetRangeSliceExample.java

```
package com.cassandraguide.rw;

//imports 已省略

public class GetRangeSliceExample {

    public static void main(String[] args) throws Exception {
        Connector conn = new Connector();
        Cassandra.Client client = conn.connect();

        System.out.println("Getting Range Slices.");

        SlicePredicate predicate = new SlicePredicate();
        List<byte[]> colNames = new ArrayList<byte[]>();
        colNames.add("a".getBytes());
        colNames.add("b".getBytes());
        predicate.column_names = colNames;

        ColumnParent parent = new ColumnParent("Standard1");

        KeyRange keyRange = new KeyRange();
        keyRange.start_key = "k1".getBytes();
        keyRange.end_key = "k2".getBytes();

        //返回一组键值切片
        List<KeySlice> results =
            client.get_range_slices(parent, predicate, keyRange,
                    ConsistencyLevel.ONE);

        for (KeySlice keySlice : results) {
            List<ColumnOrSuperColumn> cosc = keySlice.getColumns();

            System.out.println("Current row: " +
                new String(keySlice.getKey()));

            for (int i = 0; i < cosc.size(); i++) {
                Column c = cosc.get(i).getColumn();
                System.out.println(new String(c.name, "UTF-8") + " : "
                    + new String(c.value, "UTF-8"));
            }
```

```
            }

            conn.close();

            System.out.println("All done.");
        }
    }
```



这个程序假设你已经在一些行键值里有一些数据了，见 7.8 节。

程序的输出如下：

```
Getting Range Slices.
Current row: k1
a : 1
b : 2
Current row: k2
a : 2.1
b : 2.2
All done.
```

虽然名称中包含了“切片”（slice）和“区间”（range）两个词，起初看起来可能有些反常，不过实际上这并不是问题。只要记住切片指一组列，而区间指一组键值即可。在这个例子中，我们根据多个行键值取出多个列。

7.11 multiget_slice

使用 get_slice，可以根据一个指定的行键值得到一组列名，而使用 multiget_slice 则可以根据一组行键值来取出一组列，这里的行键值可以通过一个 column parent 和一个谓词来选择。也就是说，可以给定不止一个行键值，取出每个行键值对应的行里的一些列。所以，multget slice 的意思就是对应多行的多列。



曾经有个方法称为 multiget，不过现在已经推荐用 multiget_slice 来代替了。

multiget_slice 的操作方法是这样的：

```
Map<byte[],List<ColumnOrSuperColumn>> results =
     client.multiget_slice(rowKeys, parent, predicate, CL);
```

和之前一样，要指定 parent 和谓词，这里还要给出一组希望查询的行键值。这里的行键值是一个字节数组列表，每个字节数组对应于一个键值的名字。

返回的结果类型是 Map<byte[],List<ColumnOrSuperColumn>>，看起来是个复杂的数据结构，不过实际上非常简单。Map 是一个键 / 值对集合，其中作为键值的字节数列就是行键值，也就是说，在我们的例子里有两个键值，就是每个对应一个行键值。返回的映射里的每个 byte[] 键值指向一个列表，列表包含了一个或多个 ColumnOrSuperColumn 对象。使用这个结构是因为 Thrift 不支持继承。你必须知道列族的类型是 Standard 还是 Super，这样才能从中得到正确的数据对象。在我们的例子中，你可以从 ColumnOrSuperColumn 中取出 column（super_column 是空的），然后使用 column 对象读取名和值。如果需要，还能从中得到时间戳。

使用 multiget_slice 的例子位于例 7-5 之中。

例 7-5：MultigetSliceExample.java

```
package com.cassandraguide.rw;

//imports 已省略

public class MultigetSliceExample {

    private static final ConsistencyLevel CL = ConsistencyLevel.ONE;

    private static final String columnFamily = "Standard1";

    public static void main(String[] args) throws
        UnsupportedEncodingException, InvalidRequestException,
        UnavailableException, TimedOutException,
            TException, NotFoundException {

        Connector conn = new Connector();
        Cassandra.Client client = conn.connect();

        System.out.println("Running Multiget Slice.");

        SlicePredicate predicate = new SlicePredicate();
        List<byte[]> colNames = new ArrayList<byte[]>();
        colNames.add("a".getBytes());
        colNames.add("c".getBytes());
        predicate.column_names = colNames;

        ColumnParent parent = new ColumnParent(columnFamily);

        // 这里，不是指定一个行键值，而是指定一个列表
        List<byte[]> rowKeys = new ArrayList<byte[]>();
        rowKeys.add("k1".getBytes());
        rowKeys.add("k2".getBytes());

        // 返回的结果页不是一个简单的列表，而是一个映射，其中的键值是行键值
        // 值是每行对应的列数组
        Map<byte[],List<ColumnOrSuperColumn>> results =
            client.multiget_slice(rowKeys, parent, predicate, CL);

        for (byte[] key : results.keySet()) {
```

```
                List<ColumnOrSuperColumn> row = results.get(key);

                System.out.println("Row " + new String(key) + " --> ");
                for (ColumnOrSuperColumn cosc : row) {
                    Column c = cosc.column;
                    System.out.println(new String(c.name, "UTF-8") + " : "
                        + new String(c.value, "UTF-8"));
                }
            }

            conn.close();

            System.out.println("All done.");
        }
    }
```

数据库中已经有了一些键值了，我尽量保持例子的简短，以使数据的结构形象化。我们有很多在 a 和 c 之间的列，不过只对 a 和 c 两列有兴趣，所以要指定了一个切片谓词来指定 column_names 限制结果。我们还希望指定不止一行的内容，所以要使用一个字节数列表来指示要访问的行键值。

上述代码的运行结果如下所示：

```
Running Multiget Slice.
Row k2 -->
a : 2.1
Row k1 -->
a : 1
c : 3
All done.
```

可以看出，键值“k2”对应的行没有“c”列，所以 Cassandra 不会返回这列的内容。而行“k1”有“b”列，但我们的切片中并没有这样查询，所以同样不会得到。



因为我们使用了 Thrift 作为 Cassandra 的底层通信 RPC 机制，返回的结果将是无序的，所以不得不在客户端重新排序。这是因为 Thrift 不能保持顺序。multiget 实际上是多个 get 请求的包装而已。

7.12 删除

在 Cassandra 中删除数据和关系型数据库中有很大不同。在关系型数据库中，只要简单地使用 delete 语句，指定要删除的一行或多行就可以了。但在 Cassandra 之中，删除操作并不会立刻删除数据。这有个简单的原因：Cassandra 的持久化、最终一致性和分布式的设计。如果 Cassandra 直接删除数据，而此时有个节点下线了，它

将无法收到删除操作。一旦这个节点重新上线，它会错误地认为，其他收到删除操作的节点丢失了数据（因为它自己错过了删除操作，所以数据还在），它会开始修复其他节点。因此，Cassandra 需要更为复杂的机制来支持删除。这个机制称为*墓碑*（tombstone）。

墓碑是一个删除操作发起的特殊标记，作为一个占位符，用来覆盖删除的值。如果某个副本没有收到删除操作，事后墓碑可以在它再次在线的时候传播给它。这个设计的直接效果是，数据占用的空间不会在删除之后就立刻腾空。每个节点都会跟踪其上所有墓碑的年龄。一旦它们的年龄达到了 `gc_grace_seconds` 设置的时间（默认是 10 天），就会运行一次压紧操作，将墓碑垃圾回收，释放占用的磁盘空间。



记住，SSTable 是不可修改的，所以数据并没有在 SSTable 中被删除。在压紧操作中，墓碑也会被放在一起，合并后的数据是排好序的，并且会为新的合并的数据创建一个索引，新合并的、有序的、有索引的数据会被写入到一个单独的新文件中。

这里的假设是，压紧之前留下的 10 天时间足够让一个坏节点修复上线了。如果你乐意，可以把这个垃圾回收时间降的更低，来让磁盘空间更快被释放。

我们来运行一个例子，删除之前插入的数据。注意，Cassandra 中没有“delete”操作，而是 `remove`，而且它也不是真的“移除”（remove）而只是一个写（墓碑标记）操作。因为 `remove` 操作实际是墓碑的写操作，所以你还是要为这个操作指定时间戳，因为如果有多个客户端写入数据，时间戳更新的操作会胜出——这些写操作可能有墓碑，也可能有新的值。Cassandra 并不区分这些不同的写操作，只要时间戳更新就能胜出。

如下是一个简单的删除操作：

```
Connector conn = new Connector();
Cassandra.Client client = conn.connect();

String columnFamily = "Standard1";
byte[] key = "k2".getBytes(); // 行键值

Clock clock = new Clock(System.currentTimeMillis());

ColumnPath colPath = new ColumnPath();
colPath.column_family = columnFamily;
colPath.column = "b".getBytes();

client.remove(key, colPath, clock, ConsistencyLevel.ALL);
```

```
System.out.println("Remove done.");

conn.close();
```

7.13 批量变更

第 4 章里已经介绍了很多用于批量插入的批量变更（batch mutate）操作，所以这里就不再重复例子了，而只是概述一下。

要一次进行大量的插入或更新操作，可以使用 `batch_mutate` 方法，而不是 `insert` 方法。和关系型世界中的批量更新类似，`batch_mutate` 操作允许在一次调用中，对成组的很多键值进行操作，以此可以避免多次操作带来的网络开销。如果 `batch_mutate` 在一系列的更新操作中的某一个发生了失败，不会被退回，所以任何已经完成的更新操作都会继续生效。对于这种错误，客户端可以重试 `batch_mutate` 操作。



曾经有一个称为 `batch_insert` 的操作，不过已经不推荐使用了。

7.13.1 批量删除

第 4 章的应用中并没有包含任何删除操作，所以这里多介绍一些。

使用 `remove` 操作可以删除一列，而使用 `batch_mutate` 和 `Deletion` 结构可以一次进行一组复杂的删除操作。

你可以创建一个列名的列表来指示要删除的列，之后将它们间接地传给 `batch_mutate`。这里说是“间接”的，是因为你需要创建多个数据结构来运行删除。

首先，创建一个要删除的列名的列表。将它传给 `SlicePredicate`，再将 `SlicePredicate` 传给 `Deletion` 对象，然后将它传给 `Mutation` 对象，最后再将 `Mutation` 对象传给 `batch_mutate`。

下面的代码简单演示了如何进行批量删除操作。首先创建 `SlicePredicate` 对象装载要删除的列名。这里我们只要删除“b”列。然后创建 `Deleteion` 对象，里面设置这个谓词，最后创建 `Mutation` 对象，在其中设置这个 `Deletion`。

一旦设置好了 `Deletion` 对象，你就可以创建变更映射。这个映射使用字节数组键值指向所要进行删除的行。你可以使用有相同或不同 `Mutation` 对象的不同的键值进行批量删除操作。这个映射的值是一个内层的映射，这个映射本身使用要变更的列族名称字符串作为键值。而它的值则指向一个变更操作列表，是要进行的操作。

```
String columnFamily = "Standard1";
byte[] key = "k2".getBytes(); // 这是行键值

Clock clock = new Clock(System.currentTimeMillis());

SlicePredicate delPred = new SlicePredicate();
List<byte[]> delCols = new ArrayList<byte[]>();

// 这里我们指定列 "b"，其实还可以指定更多
delCols.add("b".getBytes());
delPred.column_names = delCols;

Deletion deletion = new Deletion();
deletion.predicate = delPred;
deletion.clock = clock;
Mutation mutation = new Mutation();
mutation.deletion = deletion;

Map<byte[], Map<String, List<Mutation>>> mutationMap =
    new HashMap<byte[], Map<String, List<Mutation>>>();

List<Mutation> mutationList = new ArrayList<Mutation>();
mutationList.add(mutation);

Map<String, List<Mutation>> m = new HashMap<String, List<Mutation>>();
m.put(columnFamily, mutationList);

// 就改动这个行键值，虽然我们实际可以改动更多
mutationMap.put(key, m);
client.batch_mutate(mutationMap, ConsistencyLevel.ALL);
```

还有一种方法可以用 `Deletion` 结构来指定要删除的项目：使用 `SliceRange` 替代列的 `List`，这样就可以删除一个列的区间，而不需要直接地列出列的名字了。

7.13.2 区间鬼影

你可能听人提过 Cassandra 中的“区间鬼影”（range ghost）。这是说即使你删除了一个给定行的所有列，仍然可以在区间切片中返回这个行，但列数据却是空的。这没有什么问题，只是在客户端迭代返回的数据时要注意。

7.14 编程定义keyspace和列族

现在，你也可以通过 API 来创建 keyspace 和列族了。例 7-6 示意了如何进行这个操作。

例 7-6：DefineKeyspaceExample.java

```
package com.cassandraguide.rw;

//imports 已省略
```

```
/**
 * 演示如何通过程序定义 keyspace 和列族
 */
public class DefineKeyspaceExample {

    public static void main(String[] args) throws
           UnsupportedEncodingException, InvalidRequestException,
              UnavailableException, TimedOutException,
              TException, NotFoundException, InterruptedException {

        Connector conn = new Connector();
        Cassandra.Client client = conn.connect();

        System.out.println("Defining new keyspace.");

        KsDef ksdef = new KsDef();
        ksdef.name = "ProgKS";
        ksdef.replication_factor = 1;
        ksdef.strategy_class =
            "org.apache.cassandra.locator.RackUnawareStrategy";

        List<CfDef> cfdefs = new ArrayList<CfDef>();
        CfDef cfdef1 = new CfDef();
        cfdef1.name = "ProgCF1";
        cfdef1.keyspace = ksdef.name;
        cfdefs.add(cfdef1);

        ksdef.cf_defs = cfdefs;

        client.system_add_keyspace(ksdef);

        System.out.println("Defining new cf.");
        CfDef cfdef2 = new CfDef();
        cfdef2.keyspace = ksdef.name;
        cfdef2.column_type = "Standard";
        cfdef2.name = "ProgCF";

        client.system_add_column_family(cfdef2);

        conn.close();

        System.out.println("All done.");
    }
}
```

7.15 小结

本章中，我们看到了如何使用 Cassandra 提供的丰富 API 来进行各种数据读写操作。

第 8 章 客户端

我们过去使用驱动来连接关系型数据库。比如，在 Java 里，JDBC 是抽象了不同厂商数据库的实现的 API，提供了一个使用 Statements、PreparedStatements、ResultSets 等来存取数据的一致方法。要和数据库交互，你需要一个所使用数据库的驱动。比如 Oracle、SQL Server 或是 MySQL，这些数据库的交互细节对于应用开发者来说是不可见的。使用正确的驱动，你可以使用多种编程语言与多种不同数据库进行交互。

Cassandra 有一些不同，它没有驱动。如果你决定使用 Python 和 Cassandra 交互，将无法找到一个 Cassandra 的 Python 驱动，根本就没有这种东西。与 JDBC 的那种将数据库交互从开发者的角度抽象出来有所不同，Cassandra 使用了一种完全不同的机制。Cassandra 有一个 Thrift API 和 Avro 项目提供的客户端生成层。不过，还是有一些 Cassandra 的高级客户端，它们支持的语言包括 Java、Scala、Ruby、C#、Python、Perl、PHP、C++ 以及其他语言，是由第三方的开发者为了便于自己的开发而写的。

使用这些客户端的好处是，可以更方便地将它们嵌入自己的应用中去（我们将会看到如何去做），而且它们经常会提供比基本 Thrift 接口更丰富的功能，包括连接池和 JMX 集成与监控。

在下面的章节中，我们首先会看到 Thrift 和 Avro 是如何工作的，它们如何用于 Cassandra。之后，我们会看到一些更健壮的客户端项目，它们是由独立的开发者使用不同语言写成的，提供了使用 Cassandra 数据库的多种不同选择。



如果你要写一个 Cassandra 应用，应该使用一个客户端而不要自己写底层代码。唯一的问题是如何选择一个好的客户端，可以紧跟 Cassandra 本身的更新步伐。

8.1 基本的客户端API

对于 Cassandra 0.6 或更早的版本，Thrift 是全部客户端 API 的基础。而在 0.7 版本中，Avro 开始逐渐获得支持，这是因为 Thrift 接口有一些局限，而且 Thrift 的开发也不那么积极了。比如，Thrift 有一些 Bug 已经存在了一年还没有关闭，而 Cassandra 的追随者们希望提供一个更为活跃的、吸引更多注意力的客户端层。Thrift 目前的版本是 0.2，从 2009 年开始就没有新的版本发布，文档也非常少。

在本书写作的时候，还不知道 Thrift 和 Avro 可以一起被支持多长时间，两者都在当前的代码树中。因为无法确定哪一种最终会被支持，我在这里会都介绍一些。

8.2 Thrift

Thrift 是一个驱动层接口，它提供了用于客户端使用多种语言实现的 API。Thrift 是由 Facebook 开发的，并在 2008 年捐给了 Apache 基金会，成为了一个孵化器项目。目前位于 http://incubator.apache.org/thrift，不过，如果只是为了 Cassandra 使用，你不需要单独下载 Thrift。

Thrift 是个代码生成库，支持的客户端语言包括 C++、C#、Erlang、Haskell、Java、Objective C/Cocoa、OCaml、Perl、PHP、Python、Ruby、Smalltalk 和 Squeak。 它的目标是为各种流行的语言提供便利的 RPC 调用机制，而不需要使用那些开销巨大的方式，比如 SOAP。

要使用 Thrift，要使用一个语言中立的服务定义文件，描述数据类型和服务接口。这个文件会被用作引擎的输入，为每种支持的语言生成 RPC 客户端代码库。这种静态生成的设计让它非常容易被开发者所使用，而且因为类型验证都发生在编译期而非运行期，所以代码可以很有效率地运行。



你可以阅读 Thrift 的作者写的这篇全面的论文来了解 Thrift 的实现，位于 http://incubator.apache.org/thrift/static/thrift-20070401.pdf。

Thrift 的设计提供了以下这些特性。

- 语言无关的类型
  因为类型是使用定义文件按照语言中立的方式规定的，所以它们可以被不同的语言分享。比如，C++ 的结构可以和 Python 的字典类型相互交换数据。

- 通用传输接口
  不论你使用的是磁盘文件、内存数据还是 socket 流，都可以使用同一段应用代码。

- 协议无关

  Thrift 会对数据类型进行编码和解码，可以跨协议使用。

- 支持版本

  数据类型可以加入版本信息，来支持客户端 API 的更新。

Thrift 的数据定义文件使用 .thrift 扩展名。在 Cassandra 源码目录下，有一个目录叫做 interface，其中有一个文件名为 cassandra.thrift。这个文件包含了 Cassandra 的数据定义。这里我不引用整个文件了，它看起来大致是这样的：

```
// 数据结构
struct Column {
   1: required binary name,
   2: required binary value,
   3: required i64 timestamp,
}
struct SuperColumn {
   1: required binary name,
   2: required list<Column> columns,
}

// 异常
exception NotFoundException {
}
// 其他

// 服务 API 结构
enum ConsistencyLevel {
    ZERO = 0,
    ONE = 1,
    QUORUM = 2,
    DCQUORUM = 3,
    DCQUORUMSYNC = 4,
    ALL = 5,
    ANY = 6,
}
struct SliceRange {
    1: required binary start,
    2: required binary finish,
    3: required bool reversed=0,
    4: required i32 count=100,
}
struct SlicePredicate {
    1: optional list<binary> column_names,
    2: optional SliceRange   slice_range,
}
struct KeyRange {
    1: optional string start_key,
    2: optional string end_key,
    3: optional string start_token,
    4: optional string end_token,
    5: required i32 count=100
}
```

```
//服务操作
service Cassandra {
  # auth methods
  void login(1: required string keyspace, 2:required AuthenticationRequest
      auth_request)
             throws (1:AuthenticationException authnx,
                 2:AuthorizationException authzx),

  i32 get_count(1:required string keyspace,
                2:required string key,
                3:required ColumnParent column_parent,
                4:required ConsistencyLevel consistency_level=ONE)
      throws (1:InvalidRequestException ire, 2:UnavailableException ue,
          3:TimedOutException te),
//其他

//meta-APIs
  /** 列出集群中定义的keyspace */
  set<string> describe_keyspaces(),

//其他
```

这里，我使用了一些有代表性的例子，来示意 Cassandra API 的 Thrift 定义文件的构成。通过这个文件，你可以了解 Thrift 如何写定义文件，Cassandra 客户端接口可以使用哪些操作类型，以及它们可以使用什么数据结构。

在 Cassandra 发布版中，包含 .thrift 文件的 interface 文件夹里，还有一个叫做 thrift 的文件夹。这个文件夹包含了一些子文件夹，每个都是这个 Thrift 定义生成的一种语言的绑定。

当编译 Cassandra 的时候，接下来发生的事情是这样的。Ant 工具会运行如下的目标，来生成 Java、Python 和 Perl 的绑定：

```
    <target name="gen-thrift-java">
      <echo>Generating Thrift Java code from ${basedir}/interface/
          cassandra.thrift
....</echo>
      <exec executable="thrift" dir="${basedir}
/interface">
        <arg line="--gen java" />
        <arg line="-o ${interface.thrift.dir}" />
        <arg line="cassandra.thrift" />
      </exec>
    </target>
    <target name="gen-thrift-py">
      <echo>Generating Thrift Python code from ${basedir}
/interface/cassandra.thrift ....</echo>
      <exec executable="thrift" dir="${basedir}/interface">
        <arg line="--gen py" />
        <arg line="-o ${interface.thrift.dir}" />
```

```
          <arg line="cassandra.thrift" />
        </exec>
      </target>
  // 其他
```

这些 Ant 目标会直接调用 Thrift 程序，为每种不同的语言传送不同的参数。注意，发布版中也包含有 Java API，而这些目标并不会在正常的编译中被调用。所以如果你希望使用 Perl 或是 Python 接口，那么，需要自己直接运行这些目标（或者修改 build.xml 来在编译任务中加入这些目标）。



要生成其他语言的 Thrift 绑定，可以给它传入其他的 `--gen` 开关（比如 `thrift --gen php`）。

这些 Ant 目标使用 Cassandra 的 lib 目录中的 libthrift-r917130.jar。注意，Thrift 的 JAR 包的版本号会随着 Cassandra 的更新而有变化。

8.2.1 Thrift对Java的支持

要编译 Thrift 的 Java 接口，进入到目录 <thrift-home>/lib/java。在终端里键入 `>ant` 命令来运行 build.xml 脚本。

8.2.2 异常

客户端接口可能会抛出几种你经常能看到的异常。下面列出了一些基本异常，解释了你为什么会看到这些异常，虽然有些异常不在 Thrift 定义之中。

- `AuthenticationException`
  用户使用了错误的认证信息或是用户不存在。

- `AuthorizationException`
  用户存在，但没有权限访问这个 keyspace。

- `ConfigurationException`
  当一个类装载数据库描述符时无法找到配置文件，或配置文件是错误的时候会抛出这个异常。如果你忘记为 keyspace 指定分区器或是 endpoint snitch，或者在只接受正数的配置项给出了负值或犯了其他错误，就会抛出这个异常。这个异常不通过 Thrift 接口抛出。

- `InvalidRequestException`
  用户请求的格式不正确。这可能是你向一个不存在的 keyspace 或列族请求数据，或是在请求时没有给出所有需要的参数。

- `NotFoundException`
  用户请求了一个不存在的列。

- `TException`
  当调用了一个对服务器来说不合法的 Thrift 方法时会得到这个异常。这个异常通常是在所使用的版本和服务器的版本不匹配时发生的。这个异常不通过 Thrift 接口抛出，而是 Thrift 本身的一部分。`TException` 是无法捕捉的、无法预见的异常，服务器端会弹出这个异常，然后中断当前的 Thrift 调用。它们不是你必须自己定义的应用异常。

- `TimedOutException`
  响应所用的时间超出了配置的极限，默认极限为 10 秒钟。通常这是因为服务器过载、节点故障但故障尚未被发现，或是请求的数据量非常大造成的。

- `UnavailableException`
  Cassandra 的读写过程中，响应的副本数没有达到要求的数量。这个异常不通过 Thrift 接口抛出。

8.2.3 Thrift小结

如果你希望在项目中直接使用 Thrift，那么在 Windows 上使用时有一些先决条件（参考 http://wiki.apache.org/thrift/ThriftInstallationWin32），并且很多东西都可能出错。这部分因为 Thrift 本身还太年轻，在传输的实现上有很多不足，而且自从开源以来并没有得到很多直接关注，Cassandra 可能会转向 Avro。

8.3 Avro

Apache Avro 项目是一个数据序列化，是针对 RPC 系统从 Cassandra0.7 版本开始引入的 Thrift 的替代品。Avro 由 Doug Cutting 创立，他最著名的事迹可能算是创立了 Apache Hadoop 项目和 Google MapReduce 算法的实现。

Avro 提供很多和 Thrift 以及其他数据序列化与 RPC 机制非常类似的特性，类似系统还有 Google 的 Protocol Buffer。这些特性包括：

- 强健的数据结构；
- 用于 RPC 调用的高效的、节省空间的二进制格式；
- 易于与 Python、Ruby、Smalltalk、Perl、PHP 和 Objective-C 等动态类型语言集成。

Avro 有几个 Thrift 不具备的优点，特别是应用的 RPC 不需要静态代码生成，尽管

你可以把静态代码生成作为一种对静态类型语言的优化。这个项目从某种意义上说更加成熟（当前版本号 1.3.2），而且也更加活跃。

运行 Cassandra 的 Ant 文件时，编译目标会在调用其他东西的同时，调用 `avro-generate` 目标，这就会生成 Avro 接口。这些文件和 Thrift 生成的文件一样，都放在 interface 目录中。要找到 Cassandra 的 Avro 接口的完整定义，可以查看 cassandra.avpr 文件，其中包含了 Cassandra 客户端可以使用的所有消息和操作的 JSON 定义。

```
{
  "namespace":  "org.apache.cassandra.avro",
  "protocol":   "Cassandra",

  "types": [
      {"name": "ColumnPath", "type": "record",
          "fields": [
            {"name": "column_family", "type": "string"},
            {"name": "super_column", "type": ["bytes", "null"]},
            {"name": "column", "type": ["bytes", "null"]}
        ]
      },
      {"name": "Column", "type": "record",
          "fields": [
            {"name": "name", "type": "bytes"},
            {"name": "value", "type": "bytes"},
            {"name": "timestamp", "type": "long"}
        ]
      },
      {"name": "SuperColumn", "type": "record",
          "fields": [
            {"name": "name", "type": "bytes"},
            {"name": "columns", "type": {"type": "array", "items":
      "Column"}}
        ]
      },
// 其他
      }
  ],

  "messages": {
    "get": {
        "request": [
            {"name": "keyspace", "type": "string"},
            {"name": "key", "type": "string"},
            {"name": "column_path", "type": "ColumnPath"},
            {"name": "consistency_level", "type": "ConsistencyLevel"}
        ],
        "response": "ColumnOrSuperColumn",
        "errors": ["InvalidRequestException", "NotFoundException",
            "UnavailableException", "TimedOutException"]
    },
    "insert": {
```

```
        "request": [
            {"name": "keyspace", "type": "string"},
            {"name": "key", "type": "string"},
            {"name": "column_path", "type": "ColumnPath"},
            {"name": "value", "type": "bytes"},
            {"name": "timestamp", "type": "long"},
            {"name": "consistency_level", "type": "ConsistencyLevel"}
        ],
        "response": "null",
        "errors": ["InvalidRequestException", "UnavailableException",
            "TimedOutException"]
    },
// 其他
```

JSON 格式非常简洁易懂。比如，可以看到至少有两类消息：一类表示取出请求，其他代表插入请求。每种类型的可能的异常和它们的消息打包在一起。



每种异常的含义已经在 8.2 节讨论过了。

8.3.1 Avro Ant目标

Cassandra 的 build.xml 文件定义了两个 Ant 目标，它们在 Cassandra 从源码开始编译时会被执行，使用 Cassandra 的 lib 目录中的 avro-1.2.0-dev.jar 进行代码生成。这两个目标如下所示：

```
    <!--
       生成 avro 代码
    -->
    <target name="check-avro-generate">
        <uptodate property="avroUpToDate"
                  srcfile="${interface.dir}/cassandra.avpr"
                  targetfile="${interface.avro.dir}/org/apache/cassandra/avro/
Cassandra.java" />
      <taskdef name="protocol"
               classname="org.apache.avro.specific.ProtocolTask">
        <classpath refid="cassandra.classpath" />
      </taskdef>
      <taskdef name="schema" classname="org.apache.avro.specific.
SchemaTask">
        <classpath refid="cassandra.classpath" />
      </taskdef>
      <taskdef name="paranamer"
           classname="com.thoughtworks.paranamer.ant.
ParanamerGeneratorTask">
        <classpath refid="cassandra.classpath" />
      </taskdef>
    </target>

    <target name="avro-generate" unless="avroUpToDate"
```

```
        depends="init,check-avro-generate">
  <echo>Generating avro code...</echo>
  <protocol destdir="${interface.avro.dir}">
    <fileset dir="${interface.dir}">
      <include name="**/*.avpr" />
    </fileset>
  </protocol>

  <schema destdir="${interface.avro.dir}">
    <fileset dir="${interface.dir}">
      <include name="**/*.avsc" />
    </fileset>
  </schema>
</target>
```

生成 Thrift 接口的 Ant 任务是直接调用 Thrift 可执行文件（和命令行执行是一样的），但 Avro 目标就复杂多了。`check-avro-generate` 目标使用了一个称为 `avroUpToDate` 的自定义属性。`avro-generate` 目标运行之前，必须由 `check-avro-generate` 任务先确定所有文件更新到最新版本后设置这个变量。如果生成的客户端 API 文件没有跟上当前的 schema 的更新，cassandra.avpr 文件就会被重新读入，然后生成 <CASSANDRA_HOME>/interface/avro 目录下的源代码。`org.apache.cassandra.avro.Cassandra.java` 文件表示运行时的 Java Avro 接口。

Ant 的 `<taskdef>` 标签定义了其后的目标可以运行的自定义任务。这里我们有两个自定义任务：`SchemaTask` 和 `ParanamerGeneratorTask`。`SchemaTask` 是 Avro 本身的一部分，用于针对所描述的协议生成 Java 接口和类。ParaNamer 库（paranamer-generator-2.1.jar）用于允许在运行时访问非专有的方法和构造器的参数名，通常这些名称会被编译器优化掉。它会读取 Avro 生成的 Java 类的源代码目录，输出到编译输出目录里的用于维护源码中定义的名称的 Java 类。

8.3.2 Avro规范

Avro 项目定义了一个规范，理论上说，你可以根据规范写自己的实现。Avro 支持六种复杂类型：记录（record）、枚举（enum）、数组（array）、映射（map）、联合（union）和定长（fixed）。



如果你感兴趣的话，可以阅读 http://avro.apache.org/docs/current/spec.html 的完整 Avro 规范，不过，要使用 Cassandra 并不需要这样做。

Avro 的定义使用 JavaScript 对象标记（JSON），以 schema 的方式写成。这一点和 Thrift 非常不同，在 Avro 中，当数据读取的时候，schema 总会和数据在一起。这是 Avro 的一个优点，因为这样在传输数据的时候就可以传送更少的类型信息了，

序列化也因此更加紧密和高效。因为 Avro 将数据和 schema 存放在一起，这意味着任何程序之后都可以来处理数据，独立于 RPC 机制。

8.3.3 Avro小结

在 Cassandra 0.7 版本中，Avro 是它的 RPC 和数据序列化机制。它会生成用于远程客户端与数据库交互的代码。它在业界得到支持，而且受惠于更大、也更知名的 Hadoop 项目。在可以预见的未来，它将会很好地支持 Cassandra。

关于 Avro 的更多信息，可以参考它的 Apache 项目页面：http://avro.apache.org。

8.4 Git简介

Cassandra 并不直接使用 Git，但了解一点 Git 至少对你使用那些用了 Git 的客户端有帮助。（如果你对 Git 非常熟悉，可以跳过本节。）很多开源项目最近都在向 GitHub 迁移。Git 是一个比较新的开源代码管理系统，由 Linus Torvalds 所写，用于支持他的 Linux 内核开发。它带有很多社会化的特性。GitHub 是一个 Git 项目托管站点，使用 Ruby on Rails 创建，提供免费的和商业的服务。

下面这些我们将要一一了解的客户端都在使用 Git：Web 控制台、Hector、Pelops 以及其他 Cassandra 相关的卫星项目，如 Twissandra（之前作为一个例子讨论过的，使用 Cassandra 写成的 Twitter 实现）。

要得到一个 Git 项目的代码的最简单方法是找到项目的 GitHub 主页，单击“Download Source”按钮，来下载项目主干的 .tar 或 .zip 包。



如果你在使用 Linux 发布版，比如 Ubuntu，安装 Git 非常简单。只要打开终端，输入 `>apt-get install git`，就会安装好 Git 供你使用。

如果你需要修改项目源代码（比如创建一个分支项目），就需要使用 Git 客户端。如果你使用 Windows，首先就要安装 Cygwin POSIX 模拟器，然后安装 Git。接下来，到你所感兴趣的项目的 GitHub 页面，找到项目的 Git URL。打开终端，在你准备放源码的目录使用 clone 命令。输出类似这样：

```
.../gitrep>git clone http://github.com/suguru/cassandra-webconsole.git
Initialized empty Git repository in C:/git/cassandra-webconsole/.git
remote: Counting objects: 604, done.
remote: Compressing objects: 100% (463/463), done.
remote: Total 604 (delta 248), reused 103 (delta 9)
Receiving objects: 100% (604/604), 6.24 MiB | 228 KiB/s, done.
Resolving deltas: 100% (248/248), done.
```

现在，我们有了一个以 Git 项目命名的子目录了，这样就可以编译并使用它了。关于 Git 的完整讨论超出了本书的范围，不过你可以从 GitHub.com 了解更多知识。该网站的帮助文档位于 http://help.github.com，另一个网站 http://gitref.org 也为 Git 初学者提供了不错的参考信息。

8.5　连接客户端节点

一旦设置好了集群，客户端连接到哪个节点就无关紧要了。因为 Cassandra 节点是对称的，任何节点都可以作为一个请求代理，将请求信息发送到负责数据所在区间的节点。

有一些方法可以让一切组织得更为有序和高效。

8.5.1　客户端列表

最直接的连接到集群的方法是维护一个服务器的地址或主机名的列表，并在客户端循环使用这个列表。你允许客户端从这个列表之中，按照某种规则选择连接的服务器，可能是随机选择或是顺序选择。这种方法可以看做是某种廉价的负载均衡器。

这个方法的好处是易于设置，并且不需要任何人工干预。对于测试而言，这没有问题，不过这种方法极为难以管理。

8.5.2　循环DNS

另一个可选方案是创建一个 DNS 记录来代表集群中的一组服务器。使用循环 DNS（round-robin DNS）让客户端可以简单干净地连接服务器。这个方法的优点是不需要任何维护工作，也不需要客户端逻辑来决定连接不同的节点，是推荐的方法。

8.5.3　负载均衡器

第三个方法是给 Cassandra 集群配备一个负载均衡器，配置所有客户端都连接到它。负载均衡器将作为配置扩展点。

8.6　Cassandra Web控制台

Cassandra 有一个 Web 控制台，由 Suguru Namura 提供，代码托管在 GitHub。这个控制台通过使与 Cassandra 的交互简单化，来进行各种任务和查看集群信息。在介绍你将要使用的与数据库交互的真实客户端之前，我首先介绍了这个控制台，因为它可以为你提供一个非常友好的 Cassandra 实例配置的视图。

你可以从 http://github.com/suguru/cassandra-webconsole 下载控制台，作为一个 WAR 来运行。如果你希望修改源代码，可以使用 Git 来创建一个分支仓库，或是直接从下载页下载一个打包文件即可。控制台的运行需要 Java 6 和 Tomcat 6。如果要编译这个项目，还需要 Maven 2。

让我们来简单看看它支持的特性。

- keyspace
  控制台允许你查看 keyspace 的属性，添加、重命名和删除 keyspace。也可以查看每个 keyspace 和列族的配置信息。

- 列族
  可以添加或删除列族，查看它们的键值。

- 环
  可以查看系统信息，诸如运行时间和堆内存占用等。



如果你在同一台机器上也运行了 Cassandra 实例，则需要修改 Tomcat 的端口，因为 Cassandra 把 8080 和 8084 端口用作了 JMX。

我在自己的机器上启动了一个控制台，位于 http://localhost:9999/cassandra-webconsole。当你第一次启动控制台时，会看到一个界面，让你输入需要连接的 Cassandra 服务器的信息。

分帧传输与缓存传输

有两种用于连接 Cassandra 的传输类型可供选择：分帧传输与缓存传输。分帧传输被加入到 Thrift 当中，用于支持匿名服务器。两种传输方式上没有明显的性能差异，但是客户端语言的选择可能会让你只能使用某一种。比如 Twisted Python 需要使用分帧传输，而 Haskell、Ocaml、Cocoa 和 Smalltalk（截止到本书写作时）不支持分帧传输。

因为你需要在 Cassandra 服务器上打开这一支持，控制台会请你指定使用哪种传输方式。

一旦连接到一个 Cassandra 服务器上，Web 控制台就会读取它的配置信息，并带你开始与 Cassandra 的交互。

图 8-1 显示了控制台自己的配置界面。图 8-2 展示了控制台显示的 keyspace 和列族配置信息界面的截屏。可以看到，这里有四个 keyspace。选择 Keyspace1 来显示其中的列族定义，其他的几个 keyspace 是 system、Test 和 Twitter。使用这个界面，

可以给 keyspace 加入列族、重命名 keyspace、完全删除 keyspace，或是创建新的 keyspace。

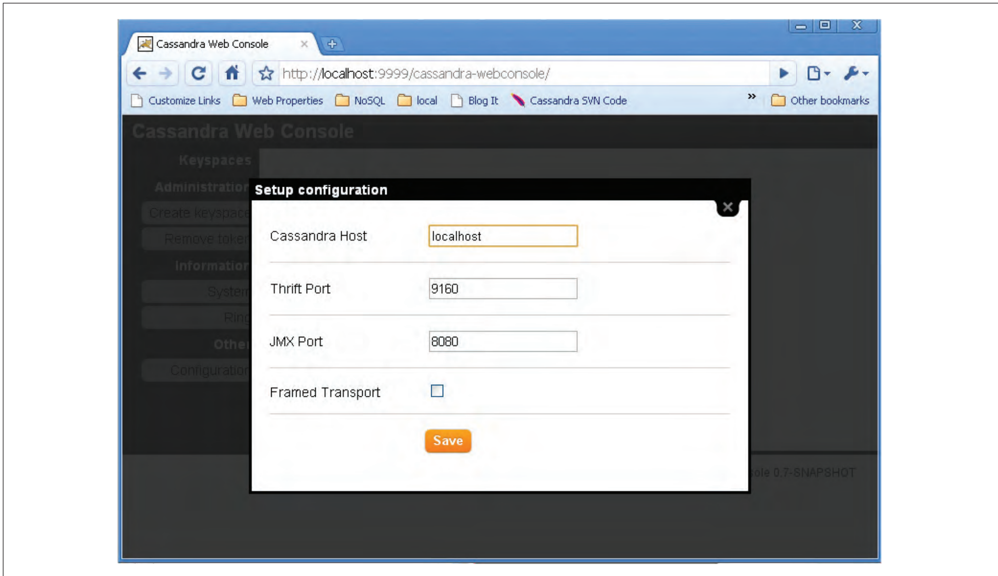


图 8-1：Cassandra Web 控制台的配置界面

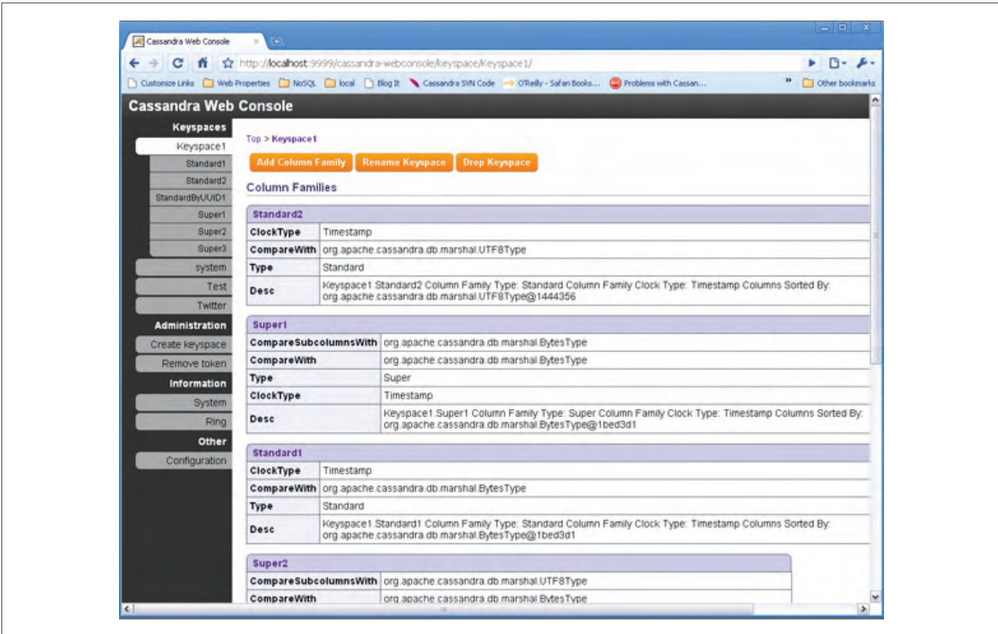


图 8-2：Web 控制台的 keyspace 和列族信息界面

如图 8-3 所示，添加一个列族或超级列族非常简单。不过，Web 控制台还不能像你所希望的那样，允许你向列族中加入数据。

图 8-3：通过 Web 控制台添加一个超级列族

如图 8-4 所示，你还可以在 Ring 界面看到服务器运行多长时间了、消耗了多少内存，以及负载情况如何。

图 8-4：Ring 界面展示了系统的使用情况

总之，Web 控制台提供了一个直观而有吸引力的界面，让 Cassandra 的基本管理任务变得更加简单了。

8.7 Hector (Java)

Hector 是一个用 Java 语言编写的，在 MIT 许可证下发布的开源项目。Hector 由 Outbrain 的 Ran Tavory（前 Google 雇员）创立，托管在 GitHub。这是 Cassandra 最早的客户端之一，并使用在 Outbrain 的产品之中。它包装了 Thrift，并提供了 JMX、连接池和故障恢复。



在希腊神话中，Hector 是特洛伊城的建造者，也是一位出色的武士。他是 Cassandra 的兄弟。

因为 Hector 是 Cassandra 的第一个客户端项目，也因为它被开发者们非常广泛地使用，甚至有其他客户端基于它开发（参考 8.8 节），我会给出一个简单而完整的使用 Hector 的例子。

要获取 Hector，可以从它的 GitHub 站点 http://github.com/rantav/hector 克隆它。如果你希望获取源代码，使用 `git` 命令即可。如果只需要二进制包，可以直接从 Download 标签下载。

8.7.1 特性

Hector 是一个很受支持的、全功能的 Cassandra 客户端，有很多用户和活跃的社区。它提供了如下功能。

- 面向对象的高级 API
  Java 开发者可以使用 Hector 提供的 Keyspace、Column 等接口，非常符合 Java 开发者的使用习惯。

- 故障恢复支持
  Thrift 不支持失败的客户端，因为 Cassandra 倾向于用在一个高度分布化的模式下，支持数据库环中有节点发生故障。但要是客户端连接到的节点刚好宕机了，如果客户端能支持故障恢复——自动寻找其他节点来完成你的请求，应该是件很不错的事。幸运的是，Hector 就提供了这个功能。

- 连接池
  Cassandra 特别为高可扩展性而设计，相应地，这也要求客户端应该支持

连接池，以便应用不会成为影响 Cassandra 性能的瓶颈。与 JDBC 一样，Cassandra 打开与关闭连接的代价也很高。Hector 的连接池使用了 Apache 的 `GenericObjectPool`。

- JMX 支持

  Cassandra 大量使用了 JMX，这对于监控来说非常方便。Hector 直接通过暴露规格，比如坏连接、可用连接、空闲连接等，来支持 JMX。

8.7.2 Hector API

下面是 Ran Tavory 的博客上（http://prettyprint.me）的用于演示 Hector 如何简化 Cassandra 使用的例子：

```
// 创建一个 cluster
Cluster c = HFactory.getOrCreateCluster("MyCluster", "cassandra1:9160");
// 选择一个 keyspace
KeyspaceOperator ko = HFactory.createKeyspaceOperator("Keyspace1", c);
// 选择一个字符串提取器
StringExtractor se = StringExtractor.get();
// 插入值
Mutator m = HFactory.createMutator(keyspaceOperator);
m.insert("key1", "ColumnFamily1", createColumn("column1", "value1", se, se));

// 现在，读一个值
// 创建一个查询
ColumnQuery<String, String> q = HFactory.createColumnQuery(keyspace-
    Operator, se, se);
// 设置键值、列名、列族名，然后执行
Result<HColumn<String, String>> r = q.setKey("key1").setName("column1").
    setColumnFamily("ColumnFamily1").execute();
// 从结果中读取值
HColumn<String, String> c = r.get();
String value =  c.getValue();
System.out.println(value);
```

8.8 HectorSharp(C#)

HectorSharp 是 Ran Tavory 的 Java 客户端 Hector 的 C# 语言移植版本（Tavory 也是 HectorSharp 项目的一个追随者）。它的特性非常类似 Hector：

- 具有直观的、面向对象接口的高级客户端；
- 客户端的故障恢复；
- 连接池；
- 负载均衡。

我们来看看如何使用 HectorSharp 作为 Cassandra 接口，如何创建一个应用。这大概是用来了解如何把它加入到项目的最好方法。我们将创建一个简单的 C# 控制台应用项目，从 Cassandra 读写一些数据，这样你就可以看到如何使用 HectorSharp 了。



在本书写作时，HectorSharp 是对应于 Cassandra 0.6 而非 0.7 的[1]。

HectorSharp 的代码可以使用 Git 从 http://github.com/mattvv/hectorsharp 获得。记住，要想方便地通过 Git 获取源代码，打开终端并进入所希望项目目录的父目录，使用 `git clone` 命令打开带有 .git 后缀的 URL，如下：

```
>git clone http://github.com/mattvv/hectorsharp.git
```

获取代码之后，需要确认安装了 .NET Framework 3.5 或更新的版本，可以从微软的官方网站免费下载它。你还可以免费使用带有 .NET Framework 4.0 的 Visual Studio .NET 2010 Express，这个集成开发环境是免费的，而且很容易用于 C# 的项目。可以从 http://www.microsoft.com/express/Downloads 下载 Visual Studio C# Express，软件的安装可能会需要一点时间，并需要重启计算机。

装好 Visual Studio 之后，打开 HectorSharp 项目，就可以看到项目的源代码，并把它加为我们自己项目的引用项目。要打开这个项目，选择 File > Open Project... 然后选择 HectorSharp.sln 文件。Express 版本的 Visual Studio 可能会声明自己不做文件解析，不过不用为此担心。

在项目浏览器窗口，右键单击 HectorSharp，从源码编译 HectorSharp。你可以在左下角看到“Build Succeeded”的提示。这样就生成了 HectorSharp.dll 文件，于是我们就可以在自己的应用中使用它了。

要创建一个使用 HectorSharp 的应用，选择 File>New Project...>Console Application。我们起名叫 ExecuteHector，你将会建立一个名为 Program.cs 带有 main 方法的 shell 类。

现在，我们要引用 HectorSharp.dll，这样才能用这个类。要做到这点，可以选择 Project > Add Reference。当对话框出现时，进入 Browse 标签页，定位到我们存放 HectorSharp 的位置，进入 bin\Release 目录，选择 HectorSharp.dll。你应该可以在项目浏览器中看到 HectorSharp 已经添加为引用库了。

译注 1：这个项目似乎从 2010 年 5 月至今都没有更新过了。



我把应用的名称修改为 CassandraProgram.cs 了。如果你也要这么做，应该通过选择 Project > ExecuteHector Properties，在项目中修改可执行文件名。选择应用标签，之后在 Startup Object 域输入你的程序名字。

让我们来简单地看一下 HectorSharp 提供的一些高级结构。

- `ICassandraClient`
  这个接口由 HectorSharp 客户端对象使用，实现类型一般为 `KeyedCassandra-ClientFactory`。

- `Pool`
  HectorSharp pool 是它到 Cassandra 的连接，可以使用一个工厂方法创建 pool，类似这样：`Pool = new CassandraClientPoolFactory().Create();`。然后使用 pool，可以创建 Client。

- `Client`
  通过连接池，你可以获取用于连接到 Cassandra 的客户端。用法非常简洁：

```
Client = new KeyedCassandraClientFactory(
             Pool,
             new KeyedCassandraClientFactory.Config { Timeout = 10 })
             .Make( new Endpoint("localhost", 9160) );
```

  首先把 pool 传入到 client 的工厂方法，然后指定附加的配置细节（比如超时的秒数），最终使用你希望连接的主机和端口得到一个端点。在上面的例子中，我们将指定 `timeout` 为 10 秒（默认为 20 秒）。

- `Keyspace`
  从 `Client` 对象获得的 Cassandra 的 keyspace。它允许你指定要连接的 keyspace 的名字和期望使用的一致性级别：

```
Keyspace = Client.GetKeyspace(
               "Keyspace1",
               ConsistencyLevel.ONE,
               new FailoverPolicy(0) { Strategy = FailoverStrategy.FAIL_FAST });
```

  `FailoverPolicy` 类允许你指定如果 HectorSharp 遇到通信错误（而非应用错误）时应该如何处理，也就是说，它是否应该认为正试图连接的节点已经宕机了。你可以重试、增量重试，或是决定退出，正如我在这里的选择。

- `ColumnPath`
  `ColumnPath` 是一个简单地包装，允许你更容易地引用整个列族、特定列族中的超级列，或是列族中的一个普通列。它只包含这三个东西的 C# 属性，以及构造器。

HectorSharp 使用了“四人帮”的 Command 模式，用于数据访问对象（DAO），因为这也是 Hector 的做法。可以通过 get 方法创建 DAO：

```
/**
 * 取一个字符串值 .
 * @return 字符串值；如果给定键的值不存在则返回 null。
 */
public String get(String key) {
  return execute(new Command<String>(){
    public String execute(Keyspace ks) {
      try {
        return string(ks.getColumn(key, createColumnPath(COLUMN_NAME)).getValue());
      } catch (NotFoundException e) {
        return null;
      }
    }
  });
}

protected static <T> T execute(Command<T> command) {
  return command.execute(CASSANDRA_HOST, CASSANDRA_PORT, CASSANDRA_KEYSPACE);
}
```

get 命令使用了参数化的 execute 方法，其他类似命令还可以用于插入和删除（例子中没有给出）。对于我们的应用来说，我们会尽量保持简单，但对于这种用例，这是一个合理的设计模式。

最终，我们已经准备就绪，可以开始写代码了。你的应用应该类似例 8-1 中的代码。

例 8-1：CassandraProgram.cs

```
using System;
using HectorSharp;
using HectorSharp.Utils;
using HectorSharp.Utils.ObjectPool;

/**
 * 一些 C# 应用将会采用 HectorSharp 作为高级 Cassandra 客户端。
 */

namespace ExecuteHector
{
    class CassandraProgram
    {
        internal ICassandraClient Client;
        internal IKeyspace Keyspace;
        internal IKeyedObjectPool<Endpoint, ICassandraClient> Pool;

        static void Main(string[] args)
        {
            CassandraProgram app = new CassandraProgram();

            Console.WriteLine("Starting HectorSharp...");
```

```
            app.Pool = new CassandraClientPoolFactory().Create();
            Console.WriteLine("Set up Pool.");
            app.Client = new KeyedCassandraClientFactory(app.Pool,
                new KeyedCassandraClientFactory.Config { Timeout = 10 })
                .Make(new Endpoint("localhost", 9160));
            Console.WriteLine("Created client.");

            app.Keyspace = app.Client.GetKeyspace(
                "Keyspace1",
                ConsistencyLevel.ONE,
                new FailoverPolicy(0) { Strategy = FailoverStrategy.
                FAIL_FAST });
            Console.WriteLine("Found keyspace " + app.Keyspace.Name);

            // 设置要使用的列路径
            var cp = new ColumnPath("Standard1", null, "MyColumn");

            // 写值
            Console.WriteLine("\nPerforming write using " +
                cp.ToString());
            for (int i = 0; i < 5; i++)
            {
                String keyname = "key" + i;
                String value = "value" + i;
                app.Keyspace.Insert(keyname, cp, value);
                Console.WriteLine("wrote to key: " + keyname  +
                    " with value: " + value);
            }
            // 读值
            Console.WriteLine("\nPerforming read.");
            for (int i = 0; i < 5; i++)
            {
                String keyname = "key" + i;
                var column = app.Keyspace.GetColumn(keyname, cp);
                Console.WriteLine("got value for " + keyname + " = " +
                    column.Value);
            }

            Console.WriteLine("All done.");
        }
    }
}
```

通过选择 Debug > Build Solution 来编译这段代码，成为一个控制台应用。

现在我们可以测试一下程序了。打开一个终端，向往常一样启动 Cassandra：`>bin\cassandra -f`。现在，再打开一个终端，进入 ExecuteHector 项目的目录，然后进入 bin\Release 目录。我们的可执行文件在这个目录里，要运行程序，只要在命令行输入 `ExecuteHector.exe` 即可。你将会看到类似下面的输出：

```
C:\git\ExecuteHector\bin\Release>ExecuteHector.exe
Starting HectorSharp...
Set up Pool.
Created client.
Found keyspace Keyspace1

Performing write using ColumnPath(family: 'Standard1', super: '', column:
    'MyColumn'
wrote to key: key0 with value: value0
wrote to key: key1 with value: value1
wrote to key: key2 with value: value2
wrote to key: key3 with value: value3
wrote to key: key4 with value: value4

Performing read.
got value for key0 = value0
got value for key1 = value1
got value for key2 = value2
got value for key3 = value3
got value for key4 = value4
All done.

C:\git\ExecuteHector\bin\Release>
```

可以看到，如果要创建一个 C# 应用，并希望使用 Cassandra 作为后端数据库，从 HectorSharp 开始会非常容易，而且他的对象模型非常高级、直观、易于使用。不过，需要当心的是，在本书写作时，HectorSharp 还非常不成熟，所以，投入大把精力之前应该先确定需求是否可以满足。

你可以在 http://hectorsharp.com 更多地了解 HectorSharp 项目。

8.9　Chirper

如果你是一个 .NET 开发者，可能会对 Chirper 感兴趣。Chirper 是 Twissandra 的一个 .NET 平台移植的版本，由 Chaker Nakhli 写成。这个项目在 Apache 2.0 许可证下发布，源代码位于 GitHub：http://github.com/nakhli/Chirper。你可以在 http://www.javageneration.com/?p=318 读到一篇介绍 Chirper 的博客。

8.10　Chiton (Python)

Chiton 是 Brandon Williams 用 Python GTK 框架写的一个 Cassandra 浏览器。你可以从 http://github.com/driftx/chiton 下载这个项目。它有一些依赖环境，所以需要一些配置才能使用。要使用 Chiton，系统需要有如下软件：

- Python 2.5 或更新的版本；
- Twisted Python（一个事件驱动的 Python 网络接口），可以在 http://twistedmatrix.com/trac 找到；

- Thrift (0.2)；
- PyGTK 2.14 或更新的版本（一个 Python 图形界面库）可以在 http://www.pygtk.org 获得。它还依赖于 GTK+，如果你使用 Linux，应该已经安装了 GTK+ 了；而如果你使用 Windows，也可以下载二进制版本的。只要将它下载到一个目录，并手工将其中的 bin 子目录加入系统路径环境变量即可。

8.11 Pelops (Java)

Pelops 是由 Dominic Williams 开发的一个免费、开源的 Java 客户端。它类似于 Hector，也是用 Java 开发的，但是这个项目更新一些。Pelops 已经成为一个很流行的客户端，它的目标包括：

- 创建一个简单、易于使用的客户端；
- 把数据处理的考虑与诸如连接池之类的底层元素分离开；
- 更紧密地跟上 Cassandra 的开发，保持最新。

Pelops 的 API 比使用 Thrift 或 Avro 暴露出来的底层 API 要简单得多。要写入数据，只需要一个 `Mutator` 类；要读数据，只需要使用 `Selector`。下面是 Williams 的网站上的一个简单例子，创建一个连接到一组 Cassandra 服务器的连接池，然后向一个超级列写入多个子列值：

```
Pelops.addPool(
    "Main",
    new String[] { "cass1.database.com", "cass2.database.com", "cass3.database.com"},
    9160,
    new Policy());

Mutator mutator = Pelops.createMutator("Main", "SupportTickets");

UuidHelper.newTimeUuidBytes(), // 使用一个时间排序的 UUID 值
mutator.newColumnList(
    mutator.newColumn("category", "videoPhone"),
    mutator.newColumn("reportType", "POOR_PICTURE"),
    mutator.newColumn("createdDate", NumberHelper.toBytes(System.
    currentTimeMillis())),
    mutator.newColumn("capture", jpegBytes),
    mutator.newColumn("comment") ));

mutator.execute(ConsistencyLevel.ONE);
```

这可比直接使用 Cassandra API 的用法简单多了。

你可以从 http://code.google.com/p/pelops 获得项目的源代码，还可以在 Dominic Williams 的网站 http://ria101.wordpress.com 看到很多关于如何使用 Pelops 的示例和解释。

如果你要在一个 Java 应用中使用 Cassandra，我建议尝试一下 Pelops。

8.12 Kundera (Java ORM)

Kundera 是一个使用 Java 注记（annotation）写成的 Cassandra 的对象关系映射实现。项目位于 http://kundera.googlecode.com，以 Apache 2.0 许可证发布。按照其作者 Impetus Labs 的介绍，Kundera 的目标是：

> ……让使用 Cassandra 变得极其简单和有趣。Kundera 不会毫无意义地重做另外一个客户端库，而是衡量已有的库并在它们之上再次封装 API，来帮助开发者避免不必要的体力劳动，只要编写简单纯粹的代码，减少代码复杂度、提升质量。最终提高生产力。

Kundera 底层使用 Pelops。一个简单的 Java 实体 bean 会类似这样：

```
@Entity
@ColumnFamily(keyspace = "Keyspace1", family = "Band")
public class Band {
    @Id
    private String id;
    @Column(name = "name")
    private String name;
    @Column(name = "instrument")
    private String instrument;
```

你可以进行一个这样的 JPA 查询：

```
Query query = entityManager.createQuery("SELECT m from Band c where
    name='george'");
List<SimpleComment> list = query.getResultList();
```

在本书写作时，这个库还相当新，看不出来是否已经一切就绪可以使用了。不过，它还是很被看好，正适合一般的应用开发者们对 Cassandra 正在迅速增长的兴趣。

8.13 Fauna (Ruby)

Twitter 的 Ryan King 以及 Evan Weaver 创建了一个 Cassandra 数据库的 Ruby 客户端，称为 Fauna。如果你从 Ruby 程序中访问 Cassandra，这个可能正是你的选择。要了解更多 Fauna 的情况，可以查看 http://github.com/fauna/cassandra/blob/master/README.rdoc。

8.14 小结

现在，你已经了解了各种 Cassandra 的客户端接口，以及如何安装使用这些客户端。有很多种 Cassandra 客户端，各有优势和局限，使用不同的语言，有不同的成熟度。这里不太可能探讨每种客户端，所以，我只选择了集中不同语言平台下的几个有代表性的客户端进行了介绍。要查看更多的可选客户端，可以访问 Cassandra 项目的 wiki 页面“客户端选择”，页面位于 http://wiki.apache.org/cassandra/ClientOptions。

第 9 章
监控

本章讨论使用各种不同的工具对 Cassandra 进行监控，并了解 Cassandra 集群生命周期中的重要事件。我们将了解一些简单的查看运行信息的方法，比如修改日志级别、理解输出等。

此外，Cassandra 还提供了内建的 Java 管理扩展（JMX）的支持，这让你对 Cassandra 节点及其底层的 Java 环境可以有更丰富的监控手段。再进行一点集成工作，我们就可以看到数据库的状态信息以及正在发生的事件，甚至是远程调节一些参数值。JMX 是 Cassandra 的一个重要部分，我们会用一些篇幅来确保我们了解 JMX 如何工作以及利用它怎样实现一些监控和管理的功能，甚至还将介绍如何写自己的 MBean 来发现新的 Cassandra 特性。让我们开始吧！

9.1 日志

最简单的了解当前数据库运行状况的方式就是修改日志级别，来输出更详细的日志信息。这对于开发和学习 Cassandra 如何运行非常有益。

Cassandra 使用 Log4J 来输出日志。默认情况下，Cassandra 服务器的日志级别是 INFO，这无法给你太多 Cassandra 正在做什么的运行细节。它只输出基本的状态更新，如下：

```
INFO 08:49:17,614 Saved Token found: 94408749511599155261361719888434486550
INFO 08:49:17,614 Saved ClusterName found: Test Cluster
INFO 08:49:17,620 Starting up server gossip
INFO 08:49:17,655 Binding thrift service to morpheus/192.168.1.5:9160
INFO 08:49:17,659 Cassandra starting up...
```

当在终端里启动 Cassandra 的时候，可以使用程序的 `-f` 参数（保持输出在终端窗口前台可见）让日志输出到这个终端窗口中来。不过，Cassandra 同时也会把日志写到物理文件中，这样你可以事后检查。

通过把日志级别改为 DEBUG，我们可以看到非常多的服务器工作的情况，而不只是状态的更新。

要修改日志级别，打开 <cassandra-home>/conf/log4j-server.properties，并在其中找到这样一行：

```
log4j.rootLogger=INFO,stdout,R
```

改成这个样子：

```
log4j.rootLogger=DEBUG,stdout,R
```



当然，在生产环境中，你可能会将日志级别调到 WARN 或是 ERROR，因为过繁琐的输出可能会让服务器变慢很多。

现在我们可以看到 Cassandra 运行的很多细节了：

```
 INFO 09:41:54,936 Completed flushing /var/lib/cassandra/data/system/
    LocationInfo-8-Data.db
DEBUG 09:41:54,942 Checking to see if compaction of LocationInfo would be
    useful
DEBUG 09:41:54,942 discard completed log segments for CommitLogContext
    (file='/var/lib/cassandra/commitlog/CommitLog-1277397714697.log',...
 INFO 09:41:54,943 Compacting [org.apache.cassandra.
    io.SSTableReader(path='/var/lib/cassandra
DEBUG 09:41:54,943 Marking replay position 121 on commit log CommitLogSegment
    (/var/lib/cassandra/commitlog/CommitLog-1277397714697.log)...
DEBUG 09:41:54,943 index size for bloom filter calc for file
    : /var/lib/cassandra/data/system/LocationInfo-5-Data.db   : 256
DEBUG 09:41:54,944 index size for bloom filter calc for file
    : /var/lib/cassandra/data/system/LocationInfo-6-Data.db   : 512
DEBUG 09:41:54,944 index size for bloom filter calc for file
    : /var/lib/cassandra/data/system/LocationInfo-7-Data.db   : 768
 INFO 09:41:54,985 Log replay complete
 INFO 09:41:55,009 Saved Token found: 94408749511599155261361719888434486550
 INFO 09:41:55,010 Saved ClusterName found: Test Cluster
 INFO 09:41:55,016 Starting up server gossip
 INFO 09:41:55,048 Binding thrift service to morpheus/192.168.1.5:9160
 INFO 09:41:55,051 Cassandra starting up...
DEBUG 09:41:55,112 Marking /var/lib/cassandra/data/system/LocationInfo-5-Data.db
    compacted
//...
DEBUG 09:41:55,117 Estimating compactions for Super1
```

```
DEBUG 09:41:55,117 Estimating compactions for Standard2
DEBUG 09:41:55,117 Estimating compactions for Super2
DEBUG 09:41:55,118 Estimating compactions for Standard1
DEBUG 09:41:55,118 Estimating compactions for StandardByUUID1
DEBUG 09:41:55,118 Estimating compactions for LocationInfo
DEBUG 09:41:55,118 Estimating compactions for HintsColumnFamily
DEBUG 09:41:55,118 Checking to see if compaction of Super1 would be useful
DEBUG 09:41:55,119 Checking to see if compaction of Standard2 would be useful
DEBUG 09:41:55,119 Checking to see if compaction of Super2 would be useful
//...
DEBUG 09:41:56,023 GC for ParNew: 1 ms, 14643776 reclaimed leaving
    80756296 used;
max is 1177812992
DEBUG 09:41:56,035 attempting to connect to lucky/192.168.1.2
DEBUG 09:41:57,025 Disseminating load info ...
```

你可以精确观察到 Cassandra 在什么时间正在做什么，这对于发现问题非常有帮助。而且，对于理解 Cassandra 如何自我维护也很有好处。

如果你希望改变日志的位置，也可以在 log4j.properties 文件里找到这行，并修改一个文件名：

```
log4j.appender.R.File=/var/log/cassandra/system.log
```

对于 Windows 来说，这个条目是一样的。在 Windows 里，这个文件将自动解析成 C:\\var\log\ cassandra\system.log。



如果在这个路径找不到日志文件，请确保你是这个目录的属主，至少拥有读写权限。如果 Cassandra 无法写日志，它不会告诉你的，只是不写而已。对于数据文件也是如此。

注意，这个位置是用于记录数据库的行为的，而非记录 Cassandra 的内部数据文件的。数据文件存放在 /var/lib/cassandra。

9.1.1 跟踪查看

要看到滚动输出的日志文件，不必使用前台开关来启动 Cassandra。你还可以不用 `-f` 选项简单地启动 Cassandra 之后，使用 `tail` 来看日志。`tail` 不是 Cassandra 专有的，在所有 Linux 发布版中都有这个实用工具，可以提供这种功能，可以在控制台看到一个文件末尾新追加的内容。

要跟踪查看日志文件，直接这样启动 Cassardra：

```
>bin/cassandra
```

然后打开第二个控制台，输入 `tail` 命令，把你希望查看的文件的路径当做参数传进去，就像这样：

```
>tail -f /var/log/cassandra/system.log
```

`-f` 参数的意思是“跟踪”（follow），随着 Cassandra 向日志文件输出信息，`tail` 也会将新的输出显示到屏幕上来。要想终止追踪显示日志，只要按下 Ctrl-C 就行了。

在 Windows 下也能做同样的事情，但是 Windows 本身并不提供 `tail` 命令。所以，要想做到这点，需要下载安装 Cygwin，这是一个 Bash Shell 模拟器。Cygwin 允许你在 Windows 下拥有 Linux 风格的界面，使用各种 Linux 工具。可以免费从 http://www.cygwin.com 获取它。

接下来就可以正常启动 Cassandra，然后使用这个命令跟踪查看日志文件了：

```
eben@lucky~$ tail -f C:\\var\\log\\cassandra\\system.log
```

这样，日志信息就会像程序在前台一样，输出到控制台了。

9.1.2 通用技巧

1. 跟踪

一旦你运行了一个打开 Debug 模式的服务器，就可以看到很多用于帮助调试的详细信息。比如，我们先写一个简单的值到数据库再读出来，可以看到日志输出是这样的：

```
DEBUG 12:55:09,778 insert
DEBUG 12:55:09,779 insert writing local key mycol
DEBUG 12:55:36,387 get
DEBUG 12:55:36,390 weakreadlocal reading SliceByNamesReadCommand(
    table='Keyspace1', key='mycol',
    columnParent='QueryPath(columnFamilyName='Standard1',
    superColumnName='null', columnName='null')',
    columns=[6b6579393939,])
```

这是在命令行执行的命令和相应的服务器输出：

```
cassandra> set Keyspace1.Standard1['mycol']['key999']='value999'
Value inserted.
cassandra> get Keyspace1.Standard1['mycol']['key999']
=> (column=6b6579393939, value=value999, timestamp=1277409309778000
```

注意这里所发生的事情。我们在名为 `Standard1` 的列族里插入了一个值。当发出 `get` 请求时，列键值 `key999` 转化成了 `6b6579393939`，因为 `Standard1` 列族的 `CompareWith` 属性为 `BytesType`。

与此不同，如果使用 `Standard2` 列族，我们将看到日志中的列名就是所输入的字

符串，因为这个列族的 CompareWith 属性是 UTF-8。这次我们写入并取出同样的值。命令行的输入如下所示：

```
cassandra> set Keyspace1.Standard2['mycol']['key888']='value888'
Value inserted.
cassandra> get Keyspace1.Standard2['mycol']['key888']
=> (column=key888, value=value888, timestamp=1277409950795000)
```

相应的服务器日志如下：

```
DEBUG 13:06:03,291 get
DEBUG 13:06:03,292 weakreadlocal reading SliceByNamesReadCommand(
    table='Keyspace1', key='mycol',
    columnParent='QueryPath(columnFamilyName='Standard2',
    superColumnName='null', columnName='null')',
    columns=[key888,])
```

通过这个例子，你应该已经充分了解了如何用日志跟踪你的动作产生的影响。

2. 危险信号

运行 Cassandra 的时候，有些事情值得留心。比如，如果你在日志里看到下面这句话，却没有其他相关信息，环上的节点可能是有什么地方出错了：

```
DEBUG 12:39:56,312 attempting to connect to mywinbox/192.168.1.3
```

尝试连接本身没什么问题，但之后应该有确认连接成功的消息。这样的消息有可能是因为在 Cassandra 集群里既有 Linux 又有 Windows 造成的，这是绝对不推荐的部署方式。如果 Linux 和 Windows 主机互相可见，并可以共享打印机等资源、互相访问文件、浏览对方提供的网页等，你可能会觉得两者共存没什么问题。但是，在生产环境里，不要试图在集群中混合搭配不同操作系统。

9.2 JMX与MBean概述

本节中，我们探索 Cassandra 如何使用 Java 管理扩展（JMX）来进行远程服务器管理。JMX 由 Java 规范请求（JSR）160 定义，并从 Java 5.0 开始成为 Java 的核心部分。



你可以通过查阅 java.lang.management 包来了解更多关于 JMX 实现的细节。

JMX 是一个 Java API，它通过两种途径提供应用管理功能。首先，JMX 允许你了解应用的健康状况和整体运行情况，如内存、线程、CPU 利用率等——这些参数对于所有 Java 应用都是适用的。其次，JMX 允许你对应用的特定方面进行稽核。

稽核（instrumentation）是指在应用代码周围进行一层封装，让应用代码提供一些钩

子（hook）给 JVM，从而允许 JVM 收集部分数据，这样外部工具就可以使用这些数据了。这些工具包括监控的代理程序、数据分析工具、性能分析工具等。JMX 不仅允许你来查看这些数据，如果应用允许，甚至可以通过更新某些值，来在运行期间管理应用。

JMX 广泛应用于各种应用的监控操作，包括：

- 检测内存不足，包括堆里的各个分度空间尺寸；
- 线程信息，诸如死锁检测、峰值线程数、当前线程数等；
- 详细的类装载信息跟踪；
- 日志级别控制；
- 通用信息，如应用运行时间、当前的 classpath。

很多流行的 Java 应用都使用 JMX 进行稽核，包括 JVM 本身、惠普的 Open View、Oracle 的 WebLogic 服务器、Glassfish 应用服务器以及 Cassandra。在这些应用里，JMX 是一种简单的管理容器的手段，而另一方面，JBoss 应用服务器使用 JMX 作为与容器交互的主要方法。

例如，WebLogic 服务通过 JMX 提供了非常广泛的行为数据。比如，监测连接池里可用的 JDBC 连接数，或是查看某个给定状态下容器中加载的无状态 bean 的个数。你不仅可以监控这些参数，还可以使用 Sun 公司（现在是 Oracle 了）JDK 提供的图形化控制台来改变它们的值。希望增加消息驱动 bean 池的尺寸？一个支持 JMX 的容器将允许你这样进行资源管理。

图 9-1 中提供了一个 JMX 架构的图示。

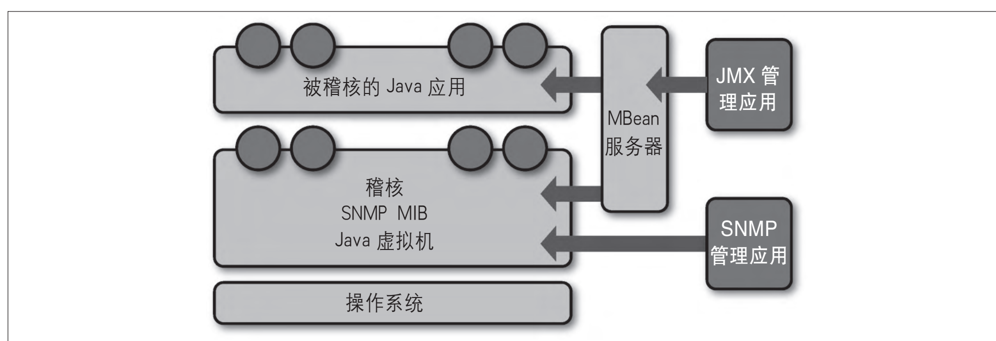


图 9-1：JMX 架构

JMX 的架构非常简单。JMX 通过底层操作系统收集信息。JMX 本身是被稽核的，所以，它的很多特性都像前面描述的那样暴露出来，用于管理了。一个被稽核的

Java 应用（比如 Cassandra）运行在 JVM 之上，也将一些特性作为可管理对象暴露出来。JDK 包含了一个 MBean 服务器，允许 JMX 管理应用通过一个远程访问协议访问应用的被稽核的特性。JVM 还提供了一些管理能力，支持简单网络监测协议（SNMP）代理使用类似方式工作。

不过，对于一个给定的程序，你只能管理应用开发者提供的可管理的东西。幸运的是，Cassandra 的开发者已经提供了对数据库的很多部分的稽核，可以非常直接地通过 JMX 进行管理。

Java 应用的稽核是通过对使用可管理 bean，对允许 JMX 加钩子的应用代码进行包装来实现的。

9.2.1 MBean

可管理 bean（managed bean），简称 MBean，是一种特殊的 Java bean，用于表示 JVM 中一个可管理的资源。MBean 与 MBean 服务器进行交互，从而支持远程管理功能。

`jconsole` 工具是标准 JDK 的自带工具。它提供了一个访问 MBean 的图形化界面，并可以支持本地和远程的管理。JConsole 的视图如图 9-2 所示。

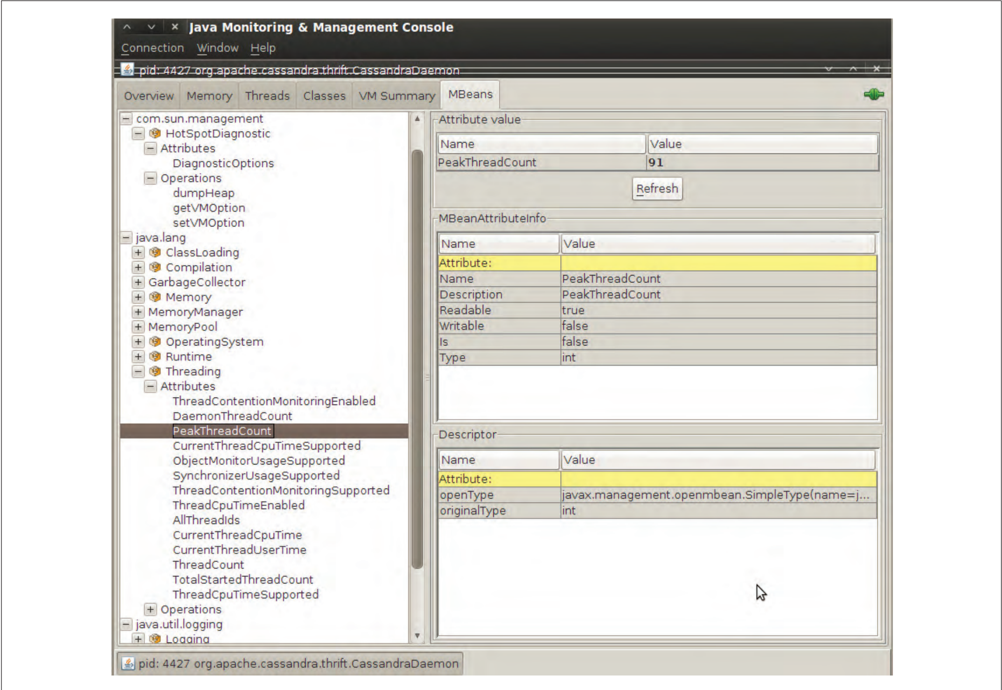


图 9-2：JConsole 显示 Cassandra 守护程序的峰值线程数

在这张图中，可以看到 JMX 提供给应用的两类视图：每个程序都有的通用的线程、内存以及 CPU 信息；一个提供了峰值线程数的更细节的视图。可以看到，应用的很多稽核内容也在这里。



JConsole 随 JDK 而发布，且易于使用，所以非常流行。不过这并不是唯一可用的 JMX 客户端。比如，JBoss 应用服务器的 Web 控制台本身就是一个 JBoss 服务器的 JMX 客户端。

应用程序或 JVM 有很多地方本来支持稽核，但是都没有打开。线程瓶颈（Thread Contention）就是一个在 JVM 中被默认关闭的有潜在用处的 MBean。这个属性可能对于调试非常有用，所以，如果你看到某个 MBean 可能会对找到问题很有用，可以打开它。不过要牢记的一点就是没有免费的午餐，阅读你要打开的 MBean 的 JavaDoc 可以帮助你了解潜在的性能影响。`ThreadCPUTime` 是另一个有用而昂贵的 MBean 的例子。

应用中一些简单的值会作为*属性*暴露出来。一个例子是 `Threading>PeakThread-Count`，这个属性直接报告 MBean 存储的应用的峰值时刻线程数。通过刷新可以查看最新的值，不过这差不多就是你能做的所有事情了。因为这个值是 JVM 内部维护的值，从外部设置它没有什么意义（这是从实际情况中记录下来的，不可配置）。

但是有的 MBean 是可以配置的。这些 MBean 会提供一些可用的操作给 JMX 代理，这样就可以获取和设置值了。可以通过查找 `writable`，来检查 MBean 是否可写。如果是 false，你就会看到一个提示，告诉你这个值是只读的；如果是 true，你就会看到一个或多个域可以设置新值，并有一个用于更新的按钮。图 9-3 给出了一个 `java.util.logging.Logger` bean 的例子。

注意，参数名对 JMX 客户端是不可见的，它们被标记为 `p0`、`p1` 之类的名字。这是因为 Java 编译器在编译阶段就已经“忘记”参数名了。所以，要了解需要设置哪些参数，需要查看相应 MBean 的 JavaDoc。对于 `java.util.logging.Logger`，这个类实现了一个称为 `java.util.logging.LoggingMXBean` 的接口，进行用于稽核的包装。要找出参数的对应关系，我们查看了这个类的 JavaDoc，发现 `p0` 是所要修改的 logger 的名字，而 `p1` 是希望设置为的日志级别。



多说一句，如果应用没有使用 `java.uitl.logging` 进行日志的话，设置这个日志级别对你不会有什么帮助。这里只是把它当做一个例子，因为它易于理解，所以就作为简单的介绍了。但是，Cassandra 没有使用这个日志包。

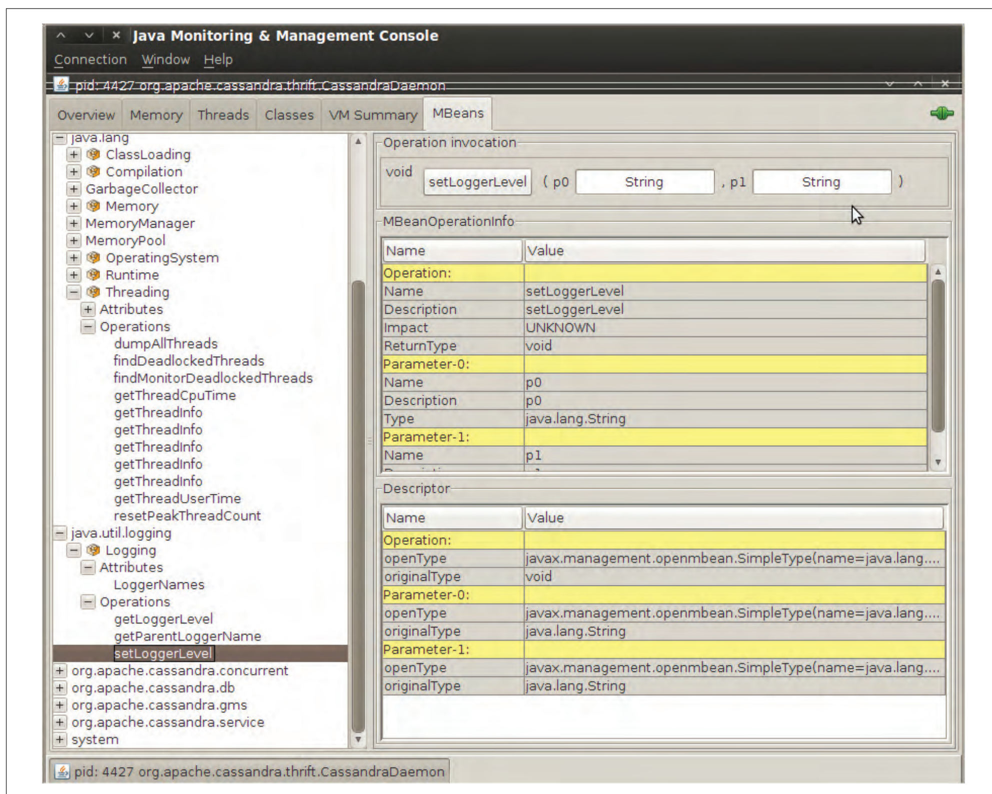


图 9-3：java.util.logging.Logger MBean 允许你设置日志级别

某些 MBean 会返回一个 `javax.management.openmbean.CompositeDataSupport` 类型的属性。这意味着，这些不是 `LoadedClassCount` 这样的简单地可以在一个域里显示的值，而是多个值。一个例子是 `Memory > HeapMemoryUsage`，它提供了很多数据点，因此有自己的视图。

另一种类型的 MBean 操作不是简单地显示一个值或允许你设置一个值，而是要执行一些有用的动作。`dumpAllThreads` 和 `resetPeakThreadCount` 就是这类操作的例子。

现在，我们很快将进入针对 Cassandra 的监控和管理。

9.2.2 集成JMX

让 Cassandra 使用 JMX 非常简单，不过还是有一些依赖的库。首先要去 http://mx4j.sourceforge.net 下载 MX4J 库 3.0.1。你没准可以看到新的可用版本，不过 3.0.1 是确实可用的。

下载 mx4j 库之后，解压并进入 lib 目录。复制其中的两个 JAR 包：mx4j.jar 和 mx4j-tools.jar。粘贴到 <cassandra-home>/lib 目录，然后重启 Cassandra。现在，其他节点就可以通过 JMX 连接并监控它的健康状态了，甚至可以通过它设置如 MBean 形式暴露出来的功能。

如果你下载了 Cassandra 的源码包，只需要直接把这两个 JAR 包放入 lib 目录，并重新编译源码。下次启动 Cassandra 的时候，你应该可以看到类似下面的输出：

```
 INFO 13:37:28,473 Cassandra starting up...
DEBUG 13:37:28,474 Will try to load mx4j now, if it's in the classpath
 INFO 13:37:28,508 mx4j successfuly loaded
HttpAdaptor version 3.0.2 started on port 8081
```

这里，MX4J 是我们的 JMX 服务器代理，现在一切已经就绪，来享用 JMX 带来的好处吧。

9.3 通过JMX与Cassandra交互

现在，JMX 支持已经打开了，我们连接到 Cassandra 的 JMX 端口上。只需要打开一个新的终端，并键入如下命令：

```
>jconsole
```

启动了 `jconsole` 后，会看到一个类似图 9-4 的登录界面。

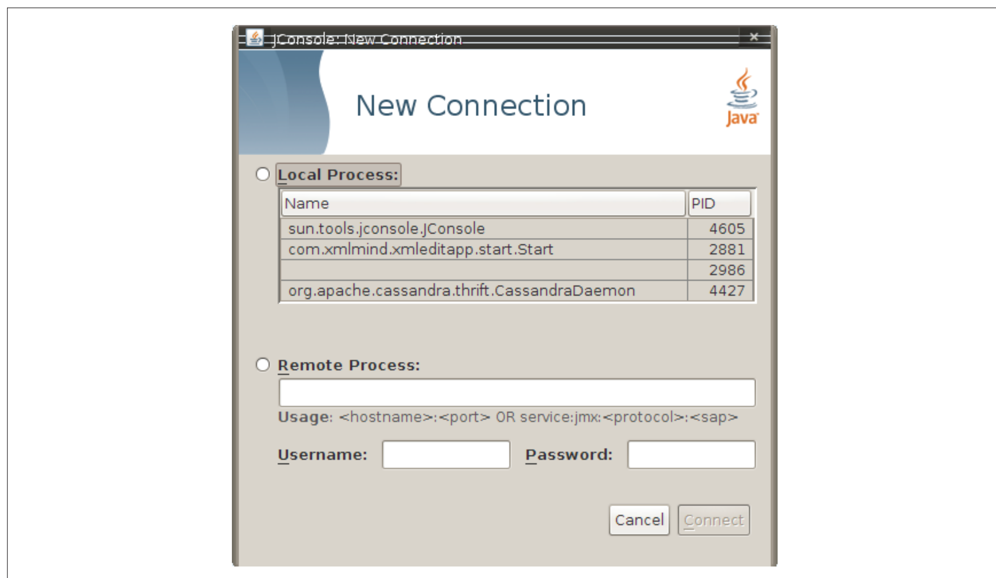


图 9-4：jconsole 登录界面

在这里，如果你要监控的就是本机，可以简单双击在本地进程部分中的 org.apache.cassandra.thrift.CassandraDaemon。如果你在监控其他节点上的 Cassandra，单击远程进程单选框，之后输入希望连接的主机和端口号。Cassandra 的 JMX 默认在 8080 端口进行广播，所以可以输入类似这样的内容，然后单击连接：

```
>lucky:8080
```



如果无法连接，要确认一下你是否有其他进程占用了 8080 端口，这个端口被很多工具使用，比如 ApacheTomcat。

一旦连接到服务器上，默认视图包含服务器状态的如下四个主要类别，它们会被不断更新。

- 堆内存
  会显示出 Cassandra 可用的总内存和正在被使用的内存。

- 线程
  会显示出 Cassandra 正在使用的活线程数量。

- 类
  Cassandra 加载的类的个数。这个数值对于一个如此强大的程序来说，是非常少的，Cassandra 通常需要使用大约 2300 个类。可以和 Oracle WebLogic 对比一下，后者通常需要加载 24 000 个类。

- CPU 利用率
  以百分比的形式显示 Cassandra 程序当前占用处理器的情况。

你可以在图表中调整显示的时间范围。

如果想要查看 Cassandra 使用 Java 堆和其他内存的细节情况，可以单击 Memory 标签。通过在下拉列表中选择合适的显示单位，你可以细致地查看 Cassandra 在其分度的内存占用情况。如果你认为必要，还可以（请求）强制垃圾回收。

可以同时连接到多个 JMX 代理。只要在菜单中选择 File > New Connection... 并重复前面的步骤，就可以连接到下一个 Cassandra 节点，同时查看多个服务器。

9.4 Cassandra的MBean

一旦你用 JConsole 或其他 JMX 代理连接上了 Cassandra，就可以用它暴露出来的 MBean 对它进行管理了。要做到这点，单击 JConsole 的 MBeans 标签。除了标准的

Java MBean，还有很多 Cassandra 包也包含了可管理 bean，按照包名排序，这些包以 `org.apache.cassandra` 开头。我们不会在这里进入到每个类的细节，但会看其中一些我们感兴趣的。

Cassandra 中的许多类都作为 MBean 暴露出来，这意味着它们实现了一个定制接口，这个接口描述了需要实现且 JMX 代理为其放置钩子的操作。写任何 MBean 的基本方式都是一样的，我会用一个简单的类作为例子。如果你希望让还不支持 JMX 的东西支持 JMX，按照这个通用的框架进行就可以了。

对于这个例子，我们来看看 Cassandra 的 `StorageService` 是如何使用 MBean 的。这里是 `StorageServiceMBean` 类的部分定义，为了简洁起见，一些操作被省略了：

```
public interface StorageServiceMBean
{
    public Set<String> getLiveNodes();

    public Set<String> getUnreachableNodes();

    public void forceTableFlush(String tableName, String... columnFamilies)
        throws IOException;

    public void removeToken(String token);

//...
}
```

可以看到，从这个 MBean 的接口定义看不出什么玄机。这只是一个常规的定义了一些操作的接口，这些操作将暴露给 JMX，`StorageService` 的实现必须支持它们。这通常意味着需要维护附加的元数据信息，用于支持这些的操作。

`StorageService` 类实现了这个接口，于是必须自己直接支持 JMX。一致性管理器域有一个 `java.util.concurrent.ExecutorService` 类型的引用，不过，实际的实现类型是 `org.apache.cassandra.concurrent.JMXEnabledThreadPool-Executor` 类型的。

```
private ExecutorService consistencyManager_ =
    new JMXEnabledThreadPoolExecutor(DatabaseDescriptor.
getConsistencyThreads(),
    DatabaseDescriptor.getConsistencyThreads(),
    StageManager.KEEPALIVE,
    TimeUnit.SECONDS,
    new LinkedBlockingQueue<Runnable>(),
    new NamedThreadFactory("CONSISTENCY-MANAGER"));
```

`JMXEnabledThreadPoolExecutor` 实现了 `JMXEnabledThreadPoolExecutor-MBean` 接口，因而也实现了 `org.apache.cassandra.concurrent.IExecutor-`

MBean，所以，所有使用线程池的 Cassandra 类都可以暴露同样的操作给 JMX。我们可以在 JMXEnabledThreadPoolExecutor 里看到 Cassandra 是如何支持 JMX 的。

执行器池在它的构造器里向平台 MBean 服务器注册自己，如下：

```
    public JMXEnabledThreadPoolExecutor(int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable> workQueue,
        NamedThreadFactory threadFactory)
    {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
threadFactory);
        super.prestartAllCoreThreads();

        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        mbeanName = "org.apache.cassandra.concurrent:type=" + threadFactory.id;
        try
        {
            mbs.registerMBean(this, new ObjectName(mbeanName));
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }
```

这里，平台服务器就是嵌入在 JDK 里的默认服务器。MBean 被命名，然后注册到平台 Mbean 服务器上，这样，MBean 就知道要监控它，并可以通过代理进行管理。

为了保持系统清洁，在 JMXEnabledThreadPoolExecutor 关闭的时候，这个类也会把自己从 MBean 服务器注销：

```
private void unregisterMBean()
{
  try
     {
     ManagementFactory.getPlatformMBeanServer().unregisterMBean
(new ObjectName(mbeanName));
     }
  catch (Exception e)
     {
     throw new RuntimeException(e);
  }
}
```

与此类似，StorageService 类也会为自己本地维护的 JMX 属性向 MBean 服务器进行注册和注销。这个类会做它本身要做的所有工作，然后再实现那些专门为了向 MBean 服务器提供信息的方法。比如，下面是 StorageService 实现的 getUnreachableNodes 操作。

```
public Set<String> getUnreachableNodes()
{
  return stringify(Gossiper.instance.getUnreachableMembers());
}
```

Gossiper 类是一个维护 IP 地址列表的单例，包含那些和本节点进行收发心跳信息的节点的地址，所以，当你在 JMX 代理中调用 getUnreachableNodes 方法时，它会调用 StorageService 的 MBean 方法，而 StorageService 则作为代理，调用 Gossiper 类的方法，后者返回不可达的 IP 地址，并包装在一个新的集合中，这样调用者就无法直接修改 Gossiper 里这个列表的内容了。

```
/* 不可达成员组 */
private Set<InetAddress> unreachableEndpoints_ =
    new ConcurrentSkipListSet<InetAddress>(inetcomparator);
//...
public Set<InetAddress> getUnreachableMembers()
{
    return new HashSet<InetAddress>(unreachableEndpoints_);
}
```

当我们在 JMX 代理（也就是 JConsole）中打开这个方法的时候，可能很欣慰地看到这个集合是空的，没有不可达节点。如果有不可达节点，它们的 IP 地址将会出现在 Value 域中，如图 9-5 所示。

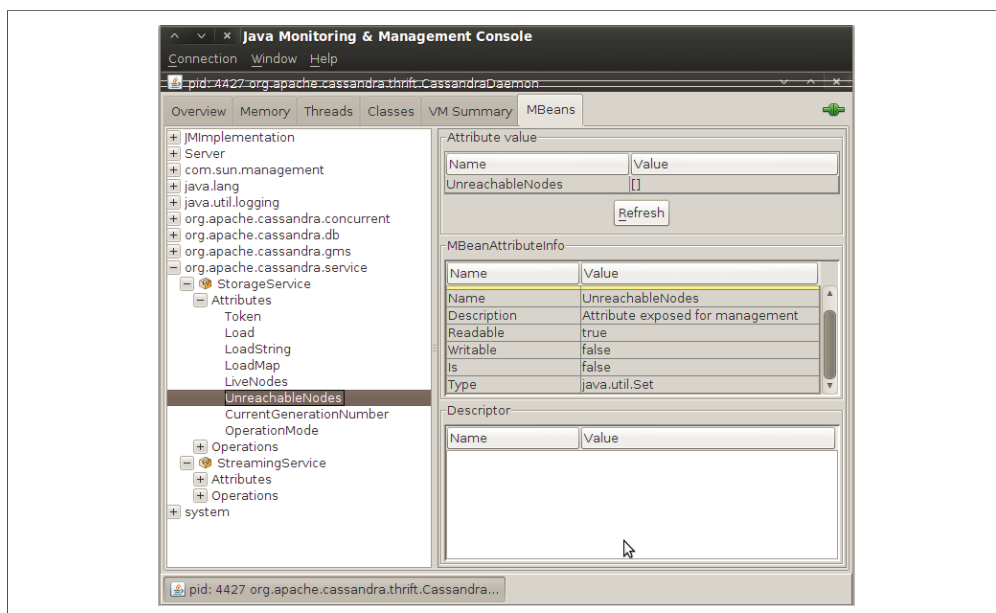


图 9-5：StorageService 的不可达节点属性

在接下来的小节中，我们将逐个包地看看可以使用 JMX 来监控和管理哪些特性。

9.4.1 org.apache.cassandra.concurrent

这个包提供了 Cassandra 的线程和流相关的功能。因而，这里有分阶段事件驱动架构（SEDA）类、gossiper、平衡器和用于刷写 memtable 中数据的类。

这些 MBean 大部分只允许你查看属性。如果你怀疑 Cassandra 环上有数据写入问题或节点不均衡的问题，可以从这里开始。

按照 SEDA 的架构，Cassandra 的每个阶段都会暴露自己的 MBean。这意味着你可以迅速判断出，在一个特定时刻处于特定阶段的对象有多少个。这些阶段包括：

- 逆熵阶段
- 迁移阶段
- 响应阶段
- 行改动阶段（用于删除和更新）
- 行读取阶段
- 流阶段

对于每个阶段，你都可以看到活动的、完成的和等待中的任务数。有些阶段，在读写数据时，维护了它们自己的线程池，而且 MBean 允许你看到线程池的尺寸。所有阶段对象都只暴露属性，不允许进行任何操作。

9.4.2 org.apache.cassandra.db

这个包里是关于 Cassandra 核心数据库本身的类，它们作为 MBean 暴露出来。这些类包括缓存、列族存储、Commit Log，以及压紧管理器。

你可以看到 Cassandra 维护的各种缓存的信息——包括列族的键和行提示的缓存——用于存放键、行和迁移的位置。缓存显示的信息包括缓存的当前尺寸、容量、缓存请求数量、缓存命中数量以及近期的缓存行为。

列族存储是另一组 MBean，它们不仅提供更丰富的属性值，也允许进行一些管理操作。它们提供 memtable、SSTable 和 Bloom filter 的使用状况信息。你可以进行的操作包括强制刷新 memtable、通过调用 `CompactionManager` 类的 `forceMajorCompaction` 方法发起主压紧，或使行缓存无效。

这里有一组关于读写数据的有用的统计信息，位于 `ColumnFamilyStores > system > Schema` bean 之中：

- 总读计数

  这是 Cassandra 所进行过的读操作的总数，可以从 ReadCount 属性得到。

- 近期读时延

  可以从 RecentReadLatencyMicros 属性读到这个数值（单位为毫秒）。还可以查看 TotalReadLatencyMicros 属性，总的读数量与读时延相乘就可以得到这个数值。

如果你发现性能正在下降，可以方便地通过这些统计信息进行分析。它们同时也展示了 Cassandra 可以做到多快。比如，在我的 8 核服务器上，在小数据库中，简单值的写时延可以达到 92 毫秒。

这组 MBean 还提供了压紧管理器的数据，包括正在进行压紧的进程中的总字节数和历史上总共被压紧的字节数。

9.4.3 org.apache.cassandra.gms

这个包有一个提供了三个操作的 MBean：转储阈值到达次数，或者获取和设置判定节点故障的阈值，默认为 8。

9.4.4 org.apache.cassandra.service

service 包提供了两个 bean：StorageService 和 StreamingService。因为这些类是 Cassandra 操作的核心，它们暴露了最多的操作，给了你相当多的外部控制手段。让我们用一些时间来研究一些核心成员。

1. StorageService

Cassandra 是一个数据库，所以它本质上是一个非常复杂的存储程序，因此，当你遇到问题的时候，应该看的第一个地方就是 StorageService MBean。这个 MBean 允许你检查 OperationMode，如果一切进行得平滑顺利，这个值应该是 normal（其他可能的状态是 leaving、joining、decommissioned 和 client）。

你还可以查看当前集群的活节点与不可达节点。如果有节点不可达，Cassandra 会在 UnreachableNodes 属性中告诉你它们的 IP 地址。这个 bean 还提供了很多其他标准的维护操作，理解这些可用操作对于保持集群处于健康状态非常重要。我建议你亲自看一下这个 bean 的代码，当然，这里我也会特别介绍一些重要的维护操作。

如果你希望在运行时改变 Cassandra 的日志级别而不中断服务（就像开始的通用例子一样），可以调用 setLog4jLevel(String classQualifier, String level)

方法。例如，为了解决某个问题，将 Cassandra 的日志级别设置为了 debug。你可以使用这里提供的一些方法来修复问题，之后，准备将系统日志级别改回到一个输出比较少的级别。这样，你需要在 JConsole 里进入到 `StorageService` MBean。我们将改变一个输出特别多的类的值：`LoadDisseminatior`。这个操作的第一个参数就是你要修改日志级别的类的名字，第二个参数是你希望设置的日志级别。输入 `org.apache.cassandra.service.LoadDisseminator` 和 `INFO`，然后单击写有 setLog4jLevel 的按钮。你将看到如下输出日志（假设日志级别已经是 debug 了）：

```
DEBUG 17:17:30,515 Disseminating load info ...
 INFO 17:17:42,001 set log level to INFO for classes under
'org.apache.cassandra.service.LoadDisseminator'
(if the level doesn't look like 'INFO' then log4j couldn't parse 'INFO')
```

从输出中可以看到，Cassandra 记录了一个加载 info 的语句，这是因为之前的日志级别是 debug。在调用 `setLog4jLevel` 操作之后，我们只得到 `INFO` 输出，而没有 debug 级别的日志输出语句了。

要了解每个节点的负载，可以使用 `getLoadMap()` 方法，这个方法返回一个 Java Map，键值为 IP 地址，值为对应节点的负载。

如果你在查找一个特定的键值，可以使用 `getNaturalEndpoints(String table, byte[] key)` 操作，传递的参数是表名和所要查找端点的键值，方法将返回一个负责存放对应键值的节点的 IP 地址列表。

你还可以使用 `getRangeToEndpointMap` 操作，获取一个区间到端点的映射，描绘出环的拓扑。

如果要废弃一个服务器，也可以使用这个 MBean。调用 `decommission()` 操作，这样当前节点的数据就会自动地传送到其他节点上，并让当前节点退出服务。

如果你足够勇敢，可以对一个给定 keyspace 中的给定列族调用 truncate 操作。如果所有节点都可用，那么这个操作将删除列族中的所有数据，只留下定义。

如果你通过其他 MBean 发现，节点已经非常不平衡了，那么进行维护的好办法是使用 `loadBalance()` 操作。不平衡随时可能发生，一些节点的数据可能会过于热门。这个操作将会导致当前的节点分出一部分数据给邻居节点，之后，它会自举并从环上负载最重的节点那里接收该节点负责的数据。

2. StreamingService

`org.cassandra.streaming.StreamingServiceMBean` 定义了如下接口：

```
public interface StreamingServiceMBean
```

```
{
    public Set<InetAddress> getStreamDestinations();

    public List<String> getOutgoingFiles(String host) throws IOException;

    public Set<InetAddress> getStreamSources();

    public List<String> getIncomingFiles(String host) throws IOException;

    public String getStatus();
}
```

这里有两个基本概念：流源头和流目的。每个节点可以将它的数据作为流发送给另一个节点，以进行负载均衡，这个类就支持这些操作。这些 MBean 方法让你看到集群中有关数据在节点间流动的一个视图。

`getStatus` 方法不是一系列可能状态的枚举值。它会返回一个字符串，形式是 `ReceivingFrom: [node] SendingTo: [node]`。

`StreamingService` MBean 与 `StorageService` MBean 经常在一起使用，如果你认为一个本该接收数据的节点没有接收数据，或者一个节点的负载不均衡甚至是宕机了，这两个服务可以在一起给你丰富的可视信息，帮助你发现集群中在某个时候到底发生了什么。

9.5 定制Cassandra的MBean

如果你希望添加一些新的支持 JMX 的特征，可以自己进行这项工作。我们来快速地写一个简单的 MBean，来看看这是如何做到的。这并不困难。

首先我们要获取源代码，并在要包装的 Cassandra 源文件旁边新建一个 MBean 接口。我们的 MBean 将会返回 Cassandra API 的当前版本，目前 Cassandra 不支持这项功能。这个 MBean 看起来就像这样：

```
package org.apache.cassandra.thrift;

public interface CassandraServerMBean {
    public String getVersion();
}
```

现在，需要打开 CassandraServer.java 的源代码，让它实现我们的 MBean 接口，并钩住 JMX 服务器。最后，我们会具体实现这个方法。Thrift 生成了一个可以从命令行界面获取的版本号，我们会直接复用它。新的 CassandraServer.java 类会成为这样：

```
package org.apache.cassandra.thrift;

import javax.management.MBeanServer;
```

```
import javax.management.ObjectName;

import org.apache.cassandra.auth.AllowAllAuthenticator;
import org.apache.cassandra.concurrent.StageManager;

//其他 import

public class CassandraServer implements Cassandra.Iface, CassandraServerMBean
{
    public static final String MBEAN_OBJECT_NAME =
        "org.apache.cassandra.service:type=CassandraService";

    public CassandraServer()
    {
        storageService = StorageService.instance;

        final MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        try
        {
            mbs.registerMBean(this, new ObjectName(MBEAN_OBJECT_NAME));
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
}

// 其他的服务器实现代码

// MBean 钩子代码
  public String getVersion() {
      String version = null;
      try {
        version = describe_version();
      } catch(TException e) {
        logger.warn("Cassandra server version unavailable: ", e.getMessage());
        version = "unavailable";
      }
      return version;
  }
}
```

我们实现了前面给出的接口，并将 Cassandra 服务器与管理平台连接到了一起。现在需要做的就是打开一个终端，进入到 <cassandra-home> 目录，使用 `$ ant` 命令编译代码，然后启动服务器。之后，运行 `jconsole`，连接到服务器实例。如图 9-6 所示，你应该可以看到这个属性已经可用了。

这里的 8.1.0 是作为一个 Thrift 常量而生成的 API 版本号，不必和 Cassandra 版本号相对应。

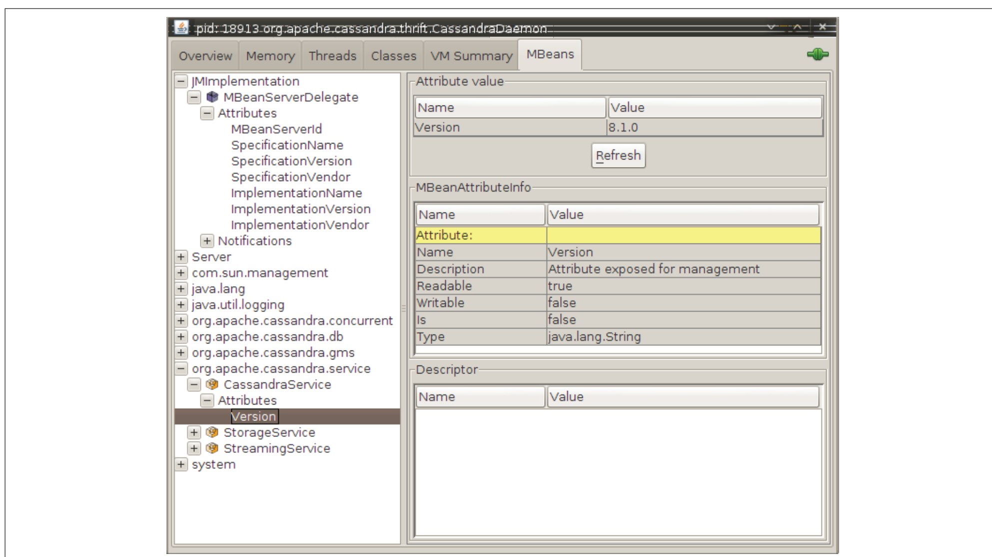


图 9-6：定制的 MBean 提供了 Cassandra 的 API 版本号



如果你在修改源代码，但是不确定修改了什么，可以通过 Subversion 来检查本地工作副本和从代码仓库中检出的版本的差异。使用 `svn diff` 命令就可以看到两个文件的变化，并可以把比较结果输出到一个新文件：

```
eben@morpheus:~$ svn diff
/home/eben/cassandra/dist/trunk-svn-1/src/java/org/apache/
cassandra/thrift/CassandraServer.java
/home/eben/cassandra/dist/trunk-svn-2/src/java/org/apache/
cassandra/thrift/CassandraServer.java
>> /home/eben/CassandraServer.patch
```

9.6 运行时分析工具

这里，我们不需要谈论太多关于 JMX 的细节，但如果要理解 Cassandra 并管理它，了解它通过 JMX 暴露出来的特性是非常有帮助的。同样，了解 Java 直接提供的一些工具也很有好处，它们可以用于发现和定位内存泄漏源、分析挂起的进程等。因为 Cassandra 是一个 Java 应用，而且还很年轻，所以现在还没有其他的专门针对 Cassandra 的第三方分析工具。



如果你正在调试，不确定使用命令行接口对 Cassandra 执行了什么命令，可以查看归属目录里的一个名为 .cassandra.history 的隐藏文件。它类似于 Bash shell 的命令历史，以纯文本的形式，按照执行顺序列出了所有执行过的命令。非常有用！

9.6.1　使用JMX和JHAT进行堆分析

在调试 Cassandra 应用的时候，经常需要了解堆上究竟发生了什么。如果你不在主干上，或者其他什么原因不太确定 Cassandra 的工作状况，这些 Java 自带工具可以帮助你。

Cassandra 会使用很多内存，而且垃圾回收和主压紧对性能有很大的冲击。这里，一个很有用的工具就是 Java 堆分析工具（JHAT）。JHAT 使用 HPROF 二进制格式创建网页，允许你浏览堆中的所有对象，查看所有堆对象的进出引用。你可以在使用这个 JDK 提供的工具的同时，从 JMX 获取其他一些可用数据。

要更好地理解堆，可以使用 `com.sun.management.HotSpotDiagnostic` bean。它提供了一个名为 `dumpHeap` 的方法。这是在 MBean 中操作的名字，所以 JMX 代理的按钮上也有这个名字。这个 bean 允许你指定一个希望把堆转储到的目标文件名，然后单击按钮，它就会将堆信息写入到你指定的文件中了。不过，`dumpHeap` 操作写出的堆状态文件是 HPROF 二进制格式的，这个格式用于堆和 CPU 的性能分析。我们不能直接看二进制文件，而需要使用 JHAT 来查看数据。

要获得堆转储，只要运行 JConsole，并找到 `com.sun.management.HotSpot-Diagnostic` bean，展开 Operations 树视图，单击 `dumpHeap` 操作。输入你希望转储到的目标文件的相对或绝对路径名，如图 9-7 所示。

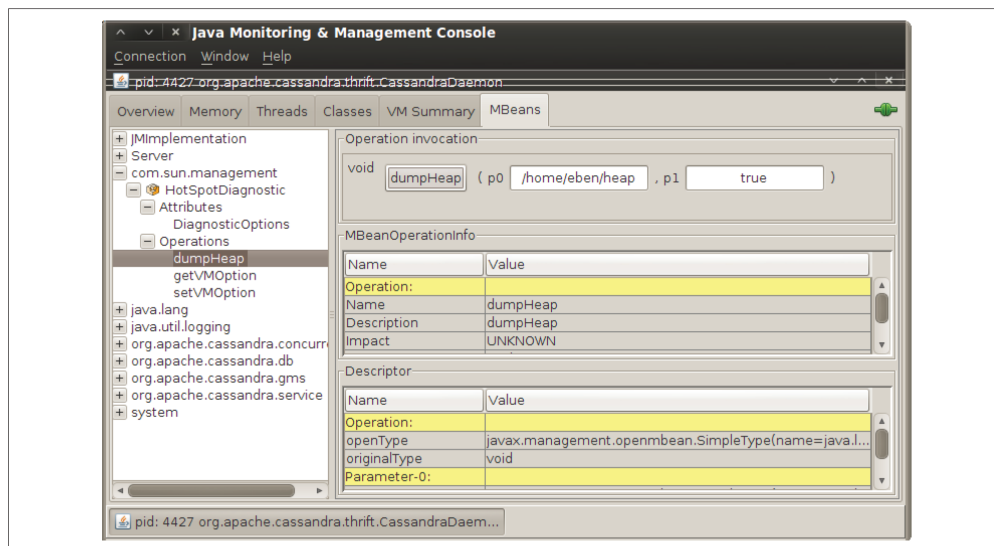


图 9-7：对 Cassandra 节点调用 dumpHeap 操作

打开一个新终端，启动 `jconsole`。你应该可以看到一个弹出框，提示说方法已经成功调用了。

你也可以使用 JDK 附带的 `jmap` 工具而不使用图形界面，获得堆转储文件。首先找到 Cassandra 进程的进程 ID。在 Linux 系统中，要获取进程的 ID 可以使用 `ps` 命令并查找‘java’。也就是，如果你有很多正在运行的 Java 进程，那么可以通过更精确地使用 `grep` 查找 Cassandra 守护程序[1]。



你可以在 Linux 系统中，打开一个终端，如下运行 `ps` 命令找到 Cassandra 的进程 ID：

```
$ ps -ef | grep 'CassandraDaemon'
```

一旦得到 Cassandra 的进程 ID，我们可以使用它来指定希望 `jmap` 工具进行快照的堆。这有助于我们找出挂起进程、定位内存泄漏源等。

```
eben@morpheus:~$ jmap -dump:live,format=b,file=/home/eben/jmapdump.bin 4427
Dumping heap to /home/eben/jmapdump.bin ...
Heap dump file created
```

现在已经获取了堆数据，让我们使用 JHAT 工具来取出它。要运行 JHAT，可以打开一个终端，输入类似如下的内容，把我的堆转出文件路径替换成你的：

```
$jhat /home/eben/jmapdump.bin
```

这个命令会运行相当一段时间，然后输出类似下面的内容：

```
Chasing references, expect 63 dots.......................................
.......................
Eliminating duplicate references.........................................
.....................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

现在，服务器已经启动了，我们可以打开浏览器，在 http://localhost:7000 查看文件。堆转储文件是可移植的，在生成堆转储之后，可以把它放到其他机器上使用 JHAT 查看。



JHAT 需要运行一阵，然后在 7000 端口启动一个 Web 服务器（JDK 6 开始，有一个集成的小型 HTTP 服务器），你可能需要关掉 JConsole 或其他可能占用这个端口的程序，因为这个端口是不可配置的。要确定端口是否可用，在 Windows 下可以使用 `>netstat -o –a`，在 Linux 下可以使用 `>netstat -o -a | less`。

要了解更多如何使用 JHAT 的信息，可以查看这个页面：http://java.sun.com/javase/6/docs/technotes/tools/share/jhat.html[2]，或通过 `jhat -h` 来查看帮助信息。

译注 1: 也可以使用 JDK 附带的 `jps` 命令，列出所有 Java 进程的 ID，会直接列出类名，比 `ps` 命令更直观。
译注 2: 由于被 Oracle 收购，页面已经转向 http://download.oracle.com/javase/6/docs/technotes/tools/share/jhat.html。

生成的网站提供了一些选项，可以限制查阅的范围：

- 所有类，包括 Java 平台的类；
- 所有类，不包括 Java 平台类；
- 根引用集（root set）的所有成员；
- 所有类的实例计数；
- 堆直方图，显示所有类和它们的创建实例数与总尺寸；
- 等待 finalizer 执行的类。

如果 Cassandra 有问题，就会创建一个 hprof 文件，同时我们可以看到如下输出：

```
DEBUG 16:13:17,640 Disseminating load info ...
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid21279.hprof ...
Heap dump file created [2766589 bytes in 0.032 secs]
ERROR 16:13:34,369 Fatal exception in thread Thread[pool-1-thread-1,5,main]
java.lang.OutOfMemoryError: Java heap space
    at org.apache.thrift.protocol.TBinaryProtocol.readStringBody(TBinaryProtocol
   .java:296)
    at org.apache.thrift.protocol.TBinaryProtocol.readMessageBegin(TBinaryProtocol
   .java:203)
    at org.apache.cassandra.thrift.Cassandra$Processor.process(Cassandra.java:1113)
```

文件将会存放在 <cassandra-home> 目录中，你可以这样打开它：

```
$ jhat java_pid21279.hprof
```

JHAT 将会分析这个文件，然后启动服务器，你就可以通过浏览器查看崩溃时所有对象的状态了。

此外，你还可以在 HTML 表单中执行一个堆数据的查询，来创建定制化视图。对象查询语言（OQL）是一种格式和 SQL 类似的用于查询堆转储文件的简单语言，可以找出和指定属性相匹配的对象。图 9-8 中的查询只允许我们查找那些属于 Cassandra 的 `org.apache.cassandra.db` 包的对象。

一旦查询返回结果，你就可以单击类名来查看关于它的详细信息。

你也可以针对更具体的对象属性进行查询。比如，你可能希望只过滤出有 200 多个字符的字符串的对象。用 OQL，可以使用下面这样的语句：

```
select s from java.lang.String s where s.count >= 200
```



关于对象查询语言的更多细节超出了本书的范围，不过它并不难使用。JHAT 工具中有很多它的例子，而且其格式与 SQL 很类似。还有一些很不错的博客文章介绍了如何使用它，特别是 A. Sundararajan 的博客，位于 http://blogs.sun.com/sundararajan/entry/querying_java_heap_with_oql。

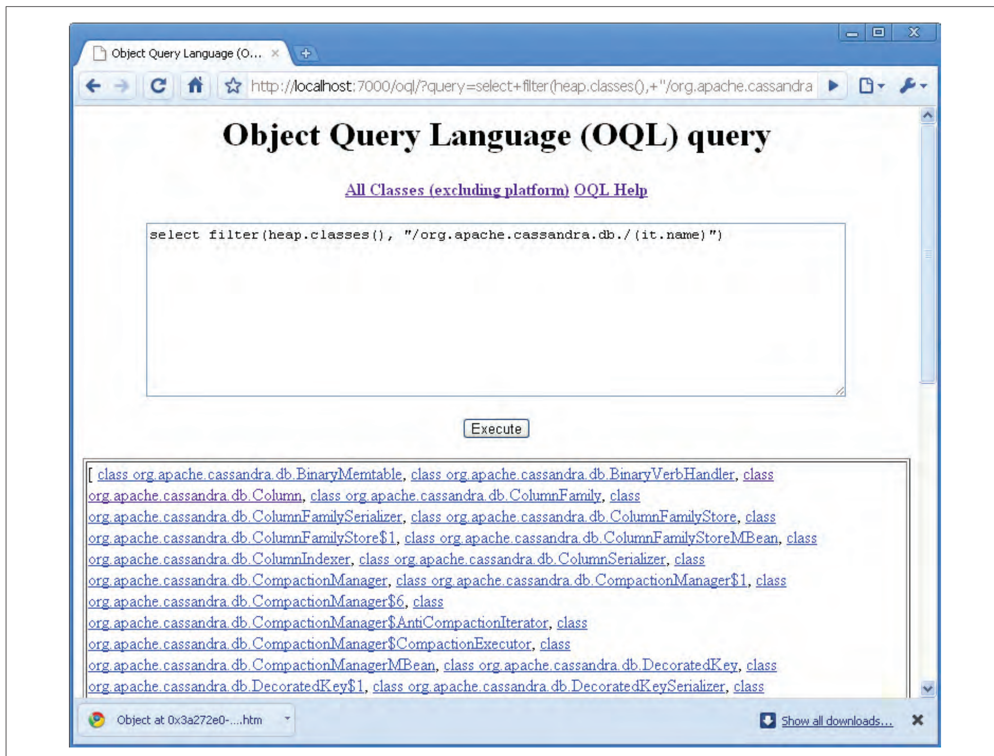


图 9-8：在 JHAT 中使用对象查询语言，过滤出 db 包中的 Cassandra 类

在这个例子里，我们可以使用查询结果来查看字符串里的 classpath 信息：

```
Object at 0x3a272e0

instance of -ea -Xdebug
-Xrunjdwp:transport=dt_socket,server=y,address=8888,suspend=n
-Xms128m -Xmx1G -XX:TargetSurvivorRatio=90 -XX:+AggressiveOpts
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled
-XX:+HeapDumpOnOutOfMemoryError
-XX:SurvivorRatio=128 -XX:MaxTenuringThreshold=0
-Dcom.sun.management.jmxremote.port=8080
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false -Dcassandra
-Dstorage-config=/opt/apache-cassandra-0.6.2/0.6.2-source/conf
-Dcassandra-foreground=yes (24 bytes)
```

这是一个迂回查看 classpath 开关的方法，不过它提供了一些关于如何使用 JMap 和 JHAT 查找你感兴趣的运行时状态的思路。

9.6.2 发现线程问题

如果你从 JConsole 发现系统变慢，或是无法控制的资源消耗增长，可能希望了解当

前的线程行为。Cassandra 大量使用了线程池和 Java 的并发库，所以，了解它们的状态是调试的一项重要工作。

在 Jconsole 中，单击 Thread 标签。左侧窗口将会显示出线程和线程池的列表，你可以单击任何一个来查看调用栈。这个视图还将显示它们的当前状态。如果你看到任何线程处于 stuck 状态，这就可能是服务器没有正常释放资源的信号，你将会围绕这个线程开始查找性能瓶颈。如果很多线程开始排队进入 stuck 的状态，服务器就将最终崩溃了。如果你正在查看异常的堆占用增长，而垃圾回收并没有释放出空间来，可能有线程 stuck 的问题，你可以检查 Jconsole 的线程部分来查看线程状态。



在 Java 中，“stuck”线程是指应用在一个超时周期后，将该线程放到了“stuck”状态，也就是说，应用已经确定，在一定时间内工作没有完成，那么这个线程将永远也无法完成它的工作。这会阻止它重新被放回到线程池中，也就是说，新的工作的可用线程会变少，也意味着性能开始受到影响。

最后，如果你关心是否有互相死锁的线程，可以单击 Detect Deadlock 按钮。

9.7 健康检查

有一些基本的检查，可以借此确保集群中的节点处于健康状态。

- 检查 `org.apache.cassandra.concurrent.ROW_MUTATION_STAGE` MBean。这里你可以看到完成任务数在增长，这意味着写操作（插入、更新和删除）在不断发生且成功完成。
- 检查 `org.apache.cassandra.concurrent.ROW_READ_STAGE` MBean。这里，也要确认完成任务计数在增长，这意味着读请求在不断进入并成功完成。
- 确认这两个 MBean 中的 `PendingTasks` 属性的值不会增长太高。即使这个数值不是很小，也应该是相对稳定的。持续增长的等待任务意味着 Cassandra 在应付负载时遇到了麻烦。和一般数据库一样，一旦问题开始了，它就会持续急剧恶化。三件事情可以改善情况：降低负载、向上扩展（升级硬件）或是水平扩展（增加节点并重新均衡）。
- 和通常一样，检查日志，并确保没有 `ERROR` 级别的日志报告。

9.8 小结

本章中，我们介绍了监控和管理 Cassandra 集群的方法。特别地，我们详细介绍了 JMX 的一些细节，学习了 Cassandra 通过 MBean 服务器提供的各种丰富操作。我们看到了如何使用 JConsole 查看 Cassandra 集群中所发生的事，并执行基本的管理任务。如果你希望的话，我们可以使用很少的 Java 代码自己使用 MBean，通过 JMX 提供新的可监控与可管理功能。

我们还比较深入地学习了如何使用 Java 开发套件（JDK）中的工具来了解 Cassandra 运行时对象的图景。使用 JMap 和 Java 堆分析工具（JHAT），我们可以对程序的内存进行快照，然后使用对象查询语言（OQL）查找具有我们关心的属性的对象。

还有很多其他更强大的维护监控工具，但这些足够帮你起步了。你的组织可能已经在使用一个可以加入钩子的工具，比如 OpenNMS（参考 http://www.opennms.org），这个工具有 JMX 钩子。又如 Nagios，这是一个开源、免费而且相当直接的工具。它有直接连接 JMX 的优势，而且这就是 Mahalo 监控 Cassandra 环的工具。



Mahalo 上有一篇文章写了它们是如何集成 Nagios 和 JMX 的，链接为 http://www.mahalo.com/how-to-monitor-cassandra-with-nagios。

现在，你应该已经准备好进行日常维护了，更好地了解了 Cassandra 集群，并知道如何进行一些常规和更细粒度的调节工作以保持 Cassandra 的健康。

第 10 章 维护

在本章中，我们会介绍一些用于保持 Cassandra 健康运行的东西。所以，带上你的安全帽，我们开始吧。

Cassandra 自带的 Nodetool 位于 <cassandra-home>/bin 目录。这是个命令行程序，提供了丰富的查看集群、了解其行为并调整集群的功能。通过 Nodetool，你可以得到有关集群的有限统计信息，查看每个节点负责的区间，在节点间移动数据、退服节点，甚至是修复遇到问题的节点。



Nodetool 的很多功能和 JMX 接口提供的功能是重合的。这是因为在台面之下，Nodetool 也是使用一个称为 `NodeProbe` 的辅助类来调用 JMX 的。所以 JMX 是进行实际工作的，`NodeProbe` 用于连接到 JMX 代理并取出数据，而 `NodeCmd` 类用于在交互式命令行界面中展现数据。

要使用 Nodetool，需要和 Cassandra 本身完全一样的环境，也就是说，它需要同样的 classpath 和日志文件。



命令行的 Nodetool 是 `org.apache.cassandra.tools.NodeCmd` 的一个包装。如果你希望确切了解它是如何工作的，可以查看该类的源代码。

启动 Nodetool 易如反掌，只要打开一个终端，进入 <cassandra-home> 目录，并执行如下命令：

```
$bin/nodetool
```

这么直接运行会得到错误输出，不过这也使程序输出一组可用的命令，下面就会介绍这些命令。

10.1 获取环的信息

你可以了解到关于环和环上节点的很多信息，这就是本节的内容。你可以获取一个单独的节点或是环上所有节点的基本信息。

10.1.1 Info

最直接的方式就是使用 `info` 命令。它告诉 Nodetool 连接一个节点，并获取关于它当前状态的基本数据。只要把要查询的节点的地址作为参数传入即可：

```
$ bin/nodetool -h 192.168.1.5 info
1344395851154532151123319526648631635 81
Load              : 3.93 MB
Generation No     : 1277663698
Uptime (seconds)  : 19639
Heap Memory (MB)  : 36.60 / 1011.25
```

这里，唯一可能有疑问的内容是“Generation No”域。这是每个端点的心跳状态，由 `Gossiper` 使用一个时间戳来维护。

10.1.2 Ring

要确定环上有哪些节点，它们状态如何，可以使用 Nodetool 的 `host` 和 `ring` 开关，如下：

```
$ bin/nodetool -host 192.168.1.5 ring
```

它会输出类似这样的结果：

```
Address       Status   Load     Range                                    Ring
                                416548800484279704830496878924242O7188
192.168.1.5   Up       1.71 KB  20846671262289044293293447172905883342   |<--|
192.168.1.7   Up       2.93 KB  41654880048427970483049687892424207188   |-->|
```

这里，我们可以看到环上所有节点的 IP 地址。这个例子里有两个节点，一个在 1.5，另一个在 1.7，两个节点的状态都是 up（目前可用且可以接受查询）。其中的 load 列表示每个节点保存的数据量。

区间令牌

keyspace 中的数据会划分为区间。Cassandra 为集群中的每个节点分配唯一的令牌，称为区间令牌（range token），这决定了哪个键值的主副本会放在这个节点上。使用 `ring` 开关进行查询得到的区间列就是每个节点负责的令牌。

有一个特殊的区间称为“折回区间”（wrapping range）。令牌值最低的节点在分到其令牌值以下的所有键值的同时，也拥有最大令牌值之上的令牌，也就是说，从最大的值折回到最小的值。

10.2 获取统计信息

Nodetool 还允许你收集精确到数据级的服务器状态统计信息。与 `info` 开关相比，这会给出详细得多的数据信息。有两个基本的统计信息命令：`cfstats` 和 `tpstats`，我们都会介绍到。

10.2.1 使用cfstats

要查看每个列族数据的概述信息，可以使用 `cfstats` 开关。这个命令会给出集群的性能指标。要查看列族的统计数据，执行如下 `Nodetool` 命令（当然，得把 IP 地址替换成你集群中的节点的地址）：

```
$ bin/nodetool -host 192.168.1.5 cfstats
```

这个命令会产生类似这样的输出：

```
Keyspace: Keyspace1
    Read Count: 13
    Read Latency: 0.3252307692307692 ms.
    Write Count: 3
    Write Latency: 0.13266666666666665 ms.
    Pending Tasks: 0
        Column Family: StandardByUUID1
        SSTable count: 0
        Space used (live): 0
        Space used (total): 0
        Memtable Columns Count: 0
        Memtable Data Size: 0
        Memtable Switch Count: 0
        Read Count: 0
        Read Latency: NaN ms.
        Write Count: 0
        Write Latency: NaN ms.
        Pending Tasks: 0
        Key cache capacity: 200000
        Key cache size: 0
        Key cache hit rate: NaN
        Row cache: disabled
        Compacted row minimum size: 0
        Compacted row maximum size: 0
        Compacted row mean size: 0
```

```
Column Family: Standard2
SSTable count: 1
Space used (live): 379
Space used (total): 379
Memtable Columns Count: 0
Memtable Data Size: 0
Memtable Switch Count: 1
Read Count: 13
Read Latency: 0.325 ms.
Write Count: 3
Write Latency: 0.133 ms.
Pending Tasks: 0
Key cache capacity: 1
Key cache size: 0
Key cache hit rate: NaN
Row cache: disabled
Compacted row minimum size: 0
Compacted row maximum size: 0
Compacted row mean size: 0
```

这里为了简单，只截取了部分输出，它为每个列族生成同样的统计信息。你可以查看读写时延、读写总数、缓存命中率，还可以在 Pending Tasks 计数中看到尚未完成的任务数。



如果看到了一个很大的等待任务数，特别是一个正在增长的等待任务数。这可能意味着 Cassandra 处于资源不足的状态，无法应付负载了。如果 Nodetool 检查出了很大的等待任务数，需要（像第 9 章介绍的那样）用 JMX 进行检查，或是（按照接下来的内容）执行 `tpstats` 命令，来查看哪些任务处于什么阶段。比如，如果很多任务处于 ROW-MUTATION-STAGE（写）可能意味着插入、更新或删除的请求进入速度比 Cassandra 的执行速度高了。

这里的统计数据展示了 Cassandra 到底有多快。不可否认，这里的负载非常低，不过写时延 1/10 毫秒仍然是个很快的数值了。

10.2.2 使用tpstats

`tpstats` 工具给出 Cassandra 维护的线程池的信息。Cassandra 是高并发的，并且特别为多处理器、多核的计算机优化。而且，Cassandra 内部采用了分阶段事件驱动架构（SEDA），所以，了解线程池的行为和健康状态对于 Cassandra 维护非常重要。

要获得线程池的统计信息，可以使用 `tpstats` 开关执行 Nodetool，如下：

```
$ bin/nodetool -host 192.168.1.5 tpstats
```

这回生成表示特定节点上的线程池的数据的 ASCII 文本输出：

```
Pool Name                    Active   Pending      Completed
FILEUTILS-DELETE-POOL             0         0            101
MESSAGING-SERVICE-POOL            2         4       71594081
STREAM-STAGE                      0         0              2
RESPONSE-STAGE                    0         0       38154433
ROW-READ-STAGE                    0         0          12542
LB-OPERATIONS                     0         0              0
COMMITLOG                         1         0       65070187
GMFD                              0         0        1002891
MESSAGE-DESERIALIZER-POOL         0         0      105025414
LB-TARGET                         0         0              0
CONSISTENCY-MANAGER               0         0           2079
ROW-MUTATION-STAGE                1         1       52419722
MESSAGE-STREAMING-POOL            0         0            121
LOAD-BALANCER-STAGE               0         0              0
FLUSH-SORTER-POOL                 0         0            115
MEMTABLE-POST-FLUSHER             0         0            115
COMPACTION-POOL                   0         0            364
FLUSH-WRITER-POOL                 0         0            115
HINTED-HANDOFF-POOL               0         0            154
```

你可以直接看到每个阶段里有多少操作，以及它们的状态是活动中、等待还是完成。这个输出是在写操作进行过程中捕捉到的，所以，其中显示 ROW-MUTATION-STAGE 有活动中的任务。

看到很多 0 意味着服务器进行的活动非常少，或者 Cassandra 正在进行异常的工作来跟上负载。

10.3 基本维护工作

在进行一些比较有影响的任务之前或之后，你需要进行一些其他任务。比如，只有在进行完刷写（flush）操作之后，进行快照才有意义。在本节中，我们就来看看这些基本的维护任务：修复、快照和清理（cleanup）。

10.3.1 修复

运行 `nodetool repair` 会让 Cassandra 执行一次主压紧。目标节点会计算出一个数据的 Merkle 树，然后将它与其他副本上的树进行对比。这步会确保不会落下任何可能与其他节点失去同步的信息。

在一次主压紧中（参考词汇表中的“压紧”），服务器会发起一个 TreeRequest/TreeReponse 会话，与相邻节点交换 Merkle 树。Merkle 树是列族数据的一个哈希表示。如果不同节点的树不匹配，它们就会进行重新协调（或“修复”），来确定它们应该设置为最新的数据值。树比较验证由 `org.apache.cassandra.service.`

AntiEntropyService 类负责。AntiEntropyService 实现了单例模式，并定义了静态的比较器类，用于比较两棵树。如果它发现任何差异，都会为不一致的部分发起一次修复。

Cassandra 会偶尔自动处理这样的问题，你也可以自己来发起它。



逆熵用于亚马逊的 Dynamo 中，Cassandra 根据它的模型实现了这个机制（如果你有学术上的兴趣，可以阅读 Dynamo 论文的 4.7 节）。

repair 命令与其他 Nodetool 命令不同，需要传入你要修复的指定 keyspace 的名字：

```
$ bin/nodetool repair Keyspace1 -h 192.168.1.7
```

运行完这个工具之后，你可以看到少量的服务器日志输出，类似这样：

```
DEBUG 13:34:59,683 Started deliverAllHints
 INFO 13:34:59,684 Compacting []
DEBUG 13:34:59,684 Expected bloom filter size : 128
DEBUG 13:34:59,685 Finished deliverAllHints
```

什么是 Merkle 树

Merkle 树是以它的发明者 Ralph Merkle 命名的，又称为“哈希树”。它的数据结构是一个二叉树，对于表示大数据集的简短摘要。在哈希树里，叶子节点是要进行摘要的数据块（通常是文件系统中的文件）。树中的每个父节点都是它的直接子节点的哈希，这可以紧密压紧摘要信息。

Cassandra 中，Merkle 树由 org.apache.cassandra.utilsMerkleTree 类实现。

在 Cassandra 之中，Merkle 树用于保证对等网络中节点收到的数据块是未被修改和破坏的。它们还用于加密和验证文件和数据传输的内容，同时 Merkle 树也用于 Google Wave 产品之中。



因为主压紧是个非常复杂和敏感的操作，所以这个任务不应该运行得太过频繁（不应该每天运行），只应该在低负载的时候进行。

10.3.2 刷写

数据存放于 memtable 中。要强制把 memtable 中的数据写到文件系统上的 SSTable 中，可以使用 Nodetool 的 flush 命令，如下：

```
bin/nodetool flush -h 192.168.1.1 -p 9160
```

如果检查服务器日志，可以看到类似这样的输出：

```
DEBUG 15:16:33,945 Forcing binary flush on keyspace Keyspace1, CF Standard2
DEBUG 15:16:33,945 Forcing flush on keyspace Keyspace1, CF Standard2
INFO 15:16:33,945 Standard2 has reached its threshold;
    switching in a fresh Memtable at
    CommitLogContext(file='/var/lib/cassandra/commitlog/CommitLog1277663698134
          .log', position=1390)
//...
INFO 15:16:34,104 Completed flushing /var/lib/cassandra/data/Keyspace1/
Standard2-3-Data.db
```

10.3.3 清理

有一个运行了一段时间的集群，你希望修改副本因子或副本策略。这是可能的，但有两个警告。首先，这种工作一般不应该在运行的集群中进行，所以一般需要关闭一些节点，再重新启动它们，即使已经动态更新了配置信息。其次，完成之后需要运行 Nodetool 的 `cleanup` 命令，以确保一切正常。

你可以用 `cleanup` 作为参数，对希望清理的节点运行 Nodetool：

```
$ bin/nodetool cleanup -h 192.168.1.7
```

这个命令会被执行，然后立即返回。实际上，这是在指定节点上运行一个反压紧操作。

10.4 快照

快照用于对一个节点的某些或全部 keyspace 进行复制，并将它保存到一个独立的数据库文件中去。这意味着可以把 keyspace 备份到其他地方，或者是把它们留在原地，当需要的时候再复原它们。

10.4.1 进行快照

当对一个或多个 keyspace 进行快照的时候，Cassandra 会调用 `Table` 类，它会对每个列族调用快照方法。这就是使用 Java 的文件工具，复制一份 SSTable 文件。注意，这里有一个问题，如果数据存在于 commit log 之中，它将不是快照的一部分，只有被刷写之后的数据才能成为快照的一部分，因为存储服务实际上就是进行了一次直接的文件复制。



要在创建快照之前进行刷写，参见 10.3.2 节。

进行快照可以直接使用如下命令：

```
$ bin/nodetool -h 192.168.1.5 snapshot
```

这会在服务器日志中打印出已经进行快照的提示：

```
DEBUG 14:25:15,385 Snapshot for Keyspace1 table data file
/var/lib/cassandra/data/Keyspace1/Standard2-1-Filter.db
created as /var/lib/cassandra/data/Keyspace1/snapshots/1277673915365/
Standard2-1-Filter.db

DEBUG 14:25:15,424 Snapshot for system table data file
/var/lib/cassandra/data/system/LocationInfo-9-Filter.db
created as /var/lib/cassandra/data/system/snapshots/1277673915389/
LocationInfo-9-Filter.d
```

注意，快照已经存放在已执行快照的时间戳命名的文件夹中了，其中的数据文件和 Cassandra 的常规表文件一样，带有 .db 扩展名。这里是对服务器上的所有 keyspace 都进行了快照，包括了 `system` 这个 Cassandra 的内部 keyspace。

如果你希望对一个指定的 keyspace 进行快照，可以把 keyspace 名字作为一个附加参数传给 Nodetool：

```
$ bin/nodetool -h 192.168.1.5 snapshot Keyspace1
```



你不必把快照放到其他地方去进行备份。如果节点的数据文件破坏了，你把这些文件留在 Cassandra 创建它们的地方会更容易恢复。

如果你希望恢复一个之前进行的快照，有几步工作需要做。首先关闭节点，并删除旧的 SSTable 和 commit log，然后把快照目录里的所有文件复制到常规数据目录里即可。

10.4.2 清除快照

你可以删除任何曾经做过的快照，比如你已经把它们备份到其他地方永久存储起来了。要清除快照，可以使用 Nodetool 的 `clearsnapshot` 开关。

你会看到服务器有这样的日志输出：

```
DEBUG 14:45:00,490 Disseminating load info ...
DEBUG 14:45:11,797 Removing snapshot directory
  /var/lib/cassandra/data/Keyspace1/snapshots
DEBUG 14:45:11,798 Deleting Standard2-1-Index.db
DEBUG 14:54:45,727 Deleting 1277675283388-Keyspace1

// 清除其他数据文件
```

```
DEBUG 14:54:45,728 Deleting snapshots
DEBUG 14:45:11,806 Cleared out all snapshot directories
```

注意这里的发生的事情：所有的快照都被清除了，包括存放在系统表中的这个 keysace 的信息。

10.5 对集群进行负载均衡

如果你发现集群变得不均衡了，可能是因为某个区间内插入了很多键值，可以重新分布数据来让集群重新变得均衡。

负载均衡与流

使用 `loadbalance` 开关执行 Nodetool 会将一个节点退服，并将它的令牌发给其他节点，然后再让它重新自举。`loadbalance` 命令实际就是退服和自举两步独立任务的一个封装。

如果你有很多数据，负载均衡会消耗很长的时间。可以通过用 `stream` 开关调用 Nodetool 来监控负载均衡操作。

这里我们使用 `loadbalance` 参数调用 Nodetool 来把 1.5 节点上的数据散布到其他节点上：

```
$ bin/nodetool -host 192.168.1.5 loadbalance
```

这样就开始了负载均衡过程。当这一切进行的时候，我们可以对希望检查的节点使用 `streams` 参数，来确定其他节点都是正常的：

```
eben@morpheus$ bin/nodetool streams -h 192.168.1.5
Mode: Leaving: streaming data to other nodes
Not sending any streams.
Not receiving any streams.

eben@morpheus$ bin/nodetool streams -h 192.168.1.7
Mode: Normal
Not sending any streams.
Not receiving any streams.
```

一旦负载均衡完成，我们可以从日志中看到负载均衡过程中都发生了什么。

这里可以看到，1.5 上的节点在开始负载均衡之后，开始离开集群，把它的数据发送给其他节点，然后自举自己，再开始重新接收数据。这就是负载均衡的效果。

```
DEBUG 10:46:58,727 Leaving: old token was 20846671262289044293293447172905
    883342
```

```
DEBUG 10:46:58,746 Pending ranges:
/192.168.1.7:
(41654880048427970483049687892424207188,
208466712622890442932934471729058833342]

 INFO 10:46:58,746 Leaving: sleeping 30000 for pending range setup
DEBUG 10:46:59,323 Disseminating load info ...
DEBUG 10:47:28,748 Node /192.168.1.5 ranges
[(41654880048427970483049687892424207188,20846671262289044293293447172905883342]]
DEBUG 10:47:28,749 Range
(41654880048427970483049687892424207188,20846671262289044293293447172905883342]
will be responsibility of /192.168.1.7
DEBUG 10:47:28,750 Ranges needing transfer are
[(41654880048427970483049687892424207188,20846671262289044293293447172905883342]]
 INFO 10:47:28,750 Leaving: streaming data to other nodes
DEBUG 10:47:28,753 Beginning transfer process to /192.168.1.7 for ranges
(41654880048427970483049687892424207188,20846671262289044293293447172905883342]
 INFO 10:47:28,753 Flushing memtables for Keyspace1...
 INFO 10:47:28,753 Performing anticompaction ...
DEBUG 10:47:28,754 waiting for stream aks.
 INFO 10:47:28,754 AntiCompacting [org.apache.cassandra.io.SSTableReader(
path='/var/lib/cassandra/data/Keyspace1/Standard2-1-Data.db')]
DEBUG 10:47:28,755 index size for bloom filter calc for file :
/var/lib/cassandra/data/Keyspace1/Standard2-1-Data.db : 256
DEBUG 10:47:28,755 Expected bloom filter size :128
 INFO 10:47:28,886 AntiCompacted to /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Data.db.
239/239 bytes for 1 keys.  Time: 131ms.
 INFO 10:47:28,887 AntiCompacting []
DEBUG 10:47:28,887 Expected bloom filter size : 128
 INFO 10:47:28,888 AntiCompacting []
 INFO 10:47:28,892 Stream context metadata /var/lib/cassandra/data/
 Keyspace1/stream/
Standard2-2-Index.db:55,
1 sstables./var/lib/cassandra/data/Keyspace1/stream/Standard2-2-Filter.db:325,
1 sstables./var/lib/cassandra/data/Keyspace1/stream/Standard2-2-Data.db:239
DEBUG 10:47:28,893 Adding file /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Index.db to be streamed.
DEBUG 10:47:28,893 Adding file /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Filter.db to be streamed.
DEBUG 10:47:28,893 Adding file /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Data.db to be streamed.
 INFO 10:47:28,895 Sending a stream initiate message to /192.168.1.7 ...
DEBUG 10:47:28,895 attempting to connect to /192.168.1.7
 INFO 10:47:28,895 Waiting for transfer to /192.168.1.7 to complete
DEBUG 10:47:29,309 Running  on default stage
DEBUG 10:47:29,310 Received a stream initiate done message ...
DEBUG 10:47:29,310 Streaming 55 length file /var/lib/cassandra/data/
Keyspace1/stream/
Standard2-2-Index.db ...
DEBUG 10:47:29,316 Bytes transferred 55
DEBUG 10:47:29,316 Done streaming /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Index.db
DEBUG 10:47:29,331 Running  on default stage
```

```
DEBUG 10:47:29,332 Deleting file /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Index.db after streaming
55/619 bytes.
DEBUG 10:47:29,332 Streaming 325 length file /var/lib/cassandra/data/
Keyspace1/stream/
Standard2-2-Filter.db ...
DEBUG 10:47:29,335 Bytes transferred 325
DEBUG 10:47:29,335 Done streaming /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Filter.db
DEBUG 10:47:29,345 Running  on default stage
DEBUG 10:47:29,345 Deleting file /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Filter.db after streaming
325/619 bytes.
DEBUG 10:47:29,345 Streaming 239 length file /var/lib/cassandra/data/Keyspace1/
stream/Standard2-2-Data.db ...
DEBUG 10:47:29,347 Bytes transferred 239
DEBUG 10:47:29,347 Done streaming /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Data.db
DEBUG 10:47:29,389 Running  on default stage
DEBUG 10:47:29,390 Deleting file /var/lib/cassandra/data/Keyspace1/stream/
Standard2-2-Data.db after streaming
239/619 bytes.
DEBUG 10:47:29,390 Signalling that streaming is done for /192.168.1.7
 INFO 10:47:29,390 Done with transfer to /192.168.1.7
DEBUG 10:47:29,391 stream acks all received.
DEBUG 10:47:29,392 No bootstrapping or leaving nodes -> empty pending ranges
for Keyspace1
DEBUG 10:48:59,322 Disseminating load info ...
DEBUG 10:49:01,406 Processing response on a callback from
191997@/192.168.1.7
 INFO 10:49:01,406 New token will be 134439585115453215112331952664863163581 to
assume load from /192.168.1.7
 INFO 10:49:01,407 re-bootstrapping to new token
134439585115453215112331952664863163581
 INFO 10:49:01,407 Joining: sleeping 30000 for pending range setup
 INFO 10:49:31,408 Bootstrapping
DEBUG 10:49:31,408 Beginning bootstrap process
DEBUG 10:49:31,413 Added /192.168.1.7/Keyspace1 as a bootstrap source
DEBUG 10:49:31,414 Requesting from /192.168.1.7 ranges
(41654880048427970483049687892424207188,134439585115453215112331952664863163581]
DEBUG 10:49:32,097 Running  on default stage
DEBUG 10:49:32,098 StreamInitiateVerbeHandler.doVerb STREAM_INITIATE 10579
```

我们来更进一步看看 Cassandra 如何做到负载均衡的。从日志输出中可以看到很多事情。首先，Cassandra 检查集群中的可用节点和它们的令牌区间。然后创建了一个 1.5 节点上需要流出的数据库文件列表。因为 `Standard2` 列族有数据，所以它将移动它的索引、数据和过滤器文件到其他节点（本例中是 1.7 节点）。然后 1.5 节点向 1.7 节点发出流初始消息，表示自己将要让自己的数据流到对方。再后开始传输数据。一旦 1.7 节点确认收到了数据，1.5 节点上的本地文件就被删除了，然后服务器

退服一段时间。紧接着，1.5 节点立刻回到集群中来，通过 gossip 告知集群，自己可以接收数据了。分配到新的区间令牌之后，1.5 节点开始从 1.7 节点服务器上分担一部分数据到本节点上来。

如果使用 `ring` 参数运行 Nodetool，就可以看到节点已经分配到了新的区间，它们都有了大致相等的数据量：

```
Address        Status   Load        Range                                      Ring
                                    134439585115453215112331952664863163581
192.168.1.7    Up       3.53 KB     41654880048427970483049687892424207188     |<--|
192.168.1.5    Up       2.95 KB     134439585115453215112331952664863163581    |-->|
```

10.6 退服节点

退服（decommissioning）一个节点就是将一个节点从服务中撤下来。当调用 Nodetool 的 `decommission` 命令时，实际上是在调用 Cassandra 的 `StorageService` 类的 `decommission` 操作。

假设环上有两个节点：

```
eben@morpheus$ bin/nodetool -host 192.168.1.5 ring
Address        Status   Load         Range                                      Ring
                                     134439585115453215112331952664863163581
192.168.1.7    Up       4.17 KB      41654880048427970483049687892424207188     |<--|
192.168.1.5    Up       3.59 KB      134439585115453215112331952664863163581    |-->|
```

现在希望退服 1.7 节点。可以用 `decommission` 参数运行 Nodetool，将一个节点撤出服务，就像这样：

```
$ bin/nodetool decommission -h 192.168.1.7
```

运行这个命令之后，它会按照配置等一段时间。默认等待时间是 30 秒。Nodetool 不会打印更多东西了，但服务器日志还会继续打印。在 1.5 节点上——我们要保留的那个节点，可以看到退服命令发出之后的日志输出：

```
DEBUG 13:59:00,488 Disseminating load info ...
DEBUG 13:59:40,010 Node /192.168.1.7 state leaving, token
41654880048427970483049687892424207188
DEBUG 13:59:40,022 Pending ranges:
/192.168.1.5:
(134439585115453215112331952664863163581,41654880048427970483049687892424207188]

DEBUG 14:00:00,488 Disseminating load info ...
DEBUG 14:00:10,184 Running  on default stage
DEBUG 14:00:10,184 StreamInitiateVerbeHandler.doVerb STREAM_INITIATE 4084
DEBUG 14:00:10,187 no data needed from /192.168.1.7
DEBUG 14:00:10,551 Node /192.168.1.7 state left, token
```

```
41654880048427970483049687892424207188
DEBUG 14:00:10,552 No bootstrapping or leaving nodes -> empty pending
ranges for
Keyspace1
 INFO 14:00:18,049 InetAddress /192.168.1.7 is now dead.
```

gossiper 告诉 1.5 节点，1.7 节点处于离开状态。在这个场景之下，因为数据已经均衡过了，1.7 节点不需要流出数据。接下来，1.5 节点就得到了消息：1.7 节点已经死掉了。如果这时运行 `nodetool -h <ip> ring` 命令，1.7 节点就不会出现在列表中了。

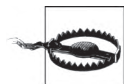
同时，在我们正在退服的节点的日志里，我们会看到这样的输出：

```
 INFO 13:37:36,299 InetAddress /192.168.1.5 is now UP
 INFO 13:59:40,929 Leaving: sleeping 30000 for pending range setup
 INFO 14:00:11,286 Leaving: streaming data to other nodes
 INFO 14:00:11,318 Flushing memtables for Keyspace1...
 INFO 14:00:11,318 Performing anticompaction ...
 INFO 14:00:11,333 AntiCompacting [org.apache.cassandra.
 io.SSTableReader(path='c
:\var\lib\cassandra\data\Keyspace1\Standard2-2-Data.db'),org.apache.cassandra.io
.SSTableReader(path='c:\var\lib\cassandra\data\Keyspace1\Standard2-1-Data.db')]
 INFO 14:00:11,349 AntiCompacting []
 INFO 14:00:11,364 AntiCompacting []
 INFO 14:00:11,411 Stream context metadata
 INFO 14:00:11,427 Sending a stream initiate message to /192.168.1.5 ...
 INFO 14:00:13,455 Shutting down MessageService...
 INFO 14:00:13,455 Shutdown complete (no further commands will be processed)
 INFO 14:00:13,470 Decommissioned
 INFO 14:00:13,470 MessagingService shutting down server thread.
```

这个节点开始发现了 1.5 节点，然后收到了退服命令。它进入离开状态，然后睡眠 30 秒，来确保有足够的时间获取所有必要的区间信息。memtable 从内存刷写到磁盘的 SSTable 之中，之后对数据进行一次反压紧操作。然后，根据需要发起一次流出数据传输，让数据流向 1.5 节点。最后，它将自己关闭，完成退服。

如果你在一个不能被退服的节点调用退服命令（比如节点还不是环的一部分，或者是唯一的可用节点），会看到一条错误消息来解释情况。执行退服命令的基本步骤是这样的。

(1) 关闭 gossiper，这样就不会收到更多的数据了。
(2) 关闭这个节点的消息服务。
(3) 关闭 SEDA 阶段管理器，因为不会接受更多的任务在阶段间移动了。
(4) 状态设置为“decommissioned”。
(5) 存储服务确定哪些可用节点对于需要传输数据的区间比较合适。将数据流到其他节点。
(6) 一旦接收节点确认数据传输成功，没有其他数据要传输了，服务器就可以离开环了。



需要当心的是，数据不会自动从退服节点上删除。如果你决定在环里重新加入先前退服的节点，负责一个新的区间，就必须首先手工删除它的数据。

10.7 更新节点

10.7.1 删除令牌

如果你想删除一个令牌，可以使用 Nodetool 来进行。

直接执行这样一条命令，其中 `removetoken` 命令的参数是将要删除的令牌：

```
$ bin/nodetool -h 127.0.0.1 removetoken 42218023250148343019074760608074740927
```

如果执行成功，客户端会无评论直接返回。注意，你不能删除一个节点自己的令牌，因为这样会破坏节点的完整性。你可以连接到节点 1 上，用它来删除环上任何地方的给定令牌。

10.7.2 压紧阈值

压紧阈值是一个数值，指一次小压紧进行之前，队列中等待被压紧的 SSTable 的数量。这个值默认为 4，最大可以设为 32。这个值不应该太小，否则 Cassandra 会经常因为过多的压紧而消耗资源，从而在很多时间都无法全力服务客户端。这个值也不能太大，这样 Cassandra 就需要花费太高的资源来进行很多压紧工作，从而没有资源用于响应客户端的请求了。

要获得一个节点当前的压紧阈值，可以使用 Nodetool 的 `getcompactionthreshold` 命令：

```
eben@morpheus$ bin/nodetool -h 192.168.1.5 getcompactionthreshold
Current compaction threshold: Min=4, Max=32
```

10.7.3 在一个工作的集群中改变列族

如果需要，可以在一个运行中的 Cassandra 集群中添加、删除或重命名列族。下面是操作的步骤。

(1) 使用 Nodetool，运行 drain 来清空 commit log。
(2) 关闭 Cassandra 并确保 commit log 中没有剩余的数据。
(3) 删除 SSTable 文件。这些文件存在于数据目录中，名为 <cf>-Data.db、<cf>-Index.db 和 <cf>-Filter.db。找到要改变的、前缀为列族名字的文件。按照对应的命名方式，重命名这些文件。

10.8 小结

本章中，我们看了一些与 Cassandra 交互，进行日常维护任务的方法。我们知道了如何获取统计信息，如何对不平衡的节点进行负载均衡，如何对数据库进行快照以备份数据，如何退服节点等。

第 11 章

性能调优

本章中，我们将学习如何对 Cassandra 进行调优以改进性能。配置文件中有很多设置可以帮助我们做到这点。我们还会介绍一些硬件选择与配置方面的优化。有一些独立的设置，你可以在 Cassandra 的配置文件中修改它们。虽然默认设置通常是推荐的，但在某些情况下，还是需要修改它们。本章中，我们将会看到这些设置。

作为一个通用的规则，必须要注意，简单直接地把一个节点加入到集群中并不会改善性能。你需要合适地分布数据副本，然后将客户端的流量分配到所有节点上。如果不将客户端请求分布化，新的节点只能一直空闲着。

我们还会看到如何使用 Cassandra 自带的 Python 压力测试工具运行一个合理的负载，来快速查看 Cassandra 在压力测试环境下的表现。之后，我们可以合理调整 Cassandra，确信已经可以在一个实战环境中启动服务。

11.1 数据存储

Cassandra 在处理更新操作时会写两类文件：commit log 和数据文件。要了解如何配置它们，需要首先明白它们的不同目的。

commit log 可以看做是短期存储。Cassandra 接收到更新之后，每个写值会立刻写入 commit log，写入是以原始顺序文件追加的形式。如果关闭了数据库，或是它意外崩溃了，commit log 可以保障数据不会丢失，因为下次启动节点的时候，commit log 还会被重放。事实上，commit log 只有在这个时候才会被读取，客户端从来不会读取它。但是，通常对 commit log 的写入操作是阻塞写，所以它可能会非常影响性能，因为它需要客户端等待它写完。

数据文件是指有序字符串表（SSTable）。和 commit log 不同，数据是异步写入这个文件的。SSTable 在周期性的主压紧中进行归并，从而释放出空间。要做到这点，Cassandra 会归并键值、合并列，并删除墓碑。

读操作可以通过内存中的缓存进行，在这种情况下不会直接读取磁盘上的数据文件。如果给 Cassandra 配置很多 GB 的内存，那么当行缓存和键缓存命中的时候，可以很大程度地提升性能。

在将所有追加写入的数据成功刷写到特定的数据文件中后，commit log 是会周期性移除的。因此，commit log 不会长到接近数据文件的尺寸，所以，它们不需要大磁盘，这在硬件选择时需要考虑。例如，如果 Cassandra 执行了一次刷写操作，可以在服务器日志中看到这样的内容：

```
INFO 18:26:11,497 Enqueuing flush of Memtable-LocationInfo@26830618(52
bytes, 2 operations)
INFO 18:26:11,497 Writing Memtable-LocationInfo@26830618(52 bytes, 2
operations)
INFO 18:26:11,732 Completed flushing /var/lib/cassandra/data/system/
LocationInfo-2-Data.db
INFO 18:26:11,732 Discarding obsolete commit log:
  CommitLogSegment(/var/lib/cassandra/commitlog/CommitLog1278894011530.log)
```

然后，如果查看 commit log 的目录，就会发现文件已经被删除了。

默认情况下，commit log 和数据文件被存放在下面的位置：

```
<CommitLogDirectory>/var/lib/cassandra/commitlog</CommitLogDirectory>

<DataFileDirectories>
  <DataFileDirectory>/var/lib/cassandra/data</DataFileDirectory>
</DataFileDirectories>
```

你可以改变这些值来改变数据文件和 commit log 存放的位置。如果你希望，可以指定多个数据文件目录。



在 Windows 下，即使它们指向默认位置，也不需要改变这些值，Windows 会自动调整路径分隔符，并把它们放在 C:\ 下面。当然，在一个真实的环境中，一般还是应该分别指定它们的路径的。

为了测试，你可能找不到要改变这些位置的理由。然而，实际部署的建议是，将数据文件和 commit log 分别放到不同的硬盘上，以使性能最大化。Cassandra 和很多其他数据库一样，特别依赖于硬盘的速度和 CPU 的速度（最好有 4 到 8 个内核，来更好发挥 Cassandra 的高并发架构优势）。确保 QA 和生产环境都安装上你能得到的

最快的硬盘，并至少有两块独立的硬盘，以便确保 commit log 文件和数据文件不会争抢 I/O 时间。而且，有多个处理器比有一两个很快的处理器要更重要。

11.2 回复超时

回复超时（reply timeout）设置的是 Cassandra 在等待其他节点响应其请求的超时失败时间。在关系型数据库和消息系统中都有这个设置。这个值由 `RpcTimeoutInMillis` 元素设置（在 YAML 中是 `rpc_timeout_in_ms`）。默认情况下，这个值是 5 000，也就是 5 秒钟。

11.3 commit log

你可以设置在停止向 commit log 追加写数据并创建新文件之前，允许它长到多大的值。这个值类似于 Log4J 的日志翻转时间设置。

这个值通过 `CommitLogRotationThresholdInMB` 元素设置（在 YAML 中是 `commitlog_rotation_threshold_in_mb`），默认值为 128 MB。

另一个 commit log 相关的设置是 sync 操作，由 `commitlog_sync` 元素设置。这个设置有两个可能的选项 `periodic` 或 `batch`。`periodic` 是默认值，它的含义是，服务器按照一个特定的时间间隔，将写操作持久化。当服务器设置为周期性持久化写的时候，有可能出现在数据还没有从滞后写缓存中同步到磁盘上的时候丢失数据。

为了确保 Cassandra 集群的持久性，可能需要修改这个设置。

如果 commit log 设置为 `batch`，它将会在数据同步写入到磁盘上之前一直阻塞写操作（Cassandra 在 commit log 完整同步到磁盘上之前，不会确认写操作）。这显然会对性能造成负面影响。

你可以把这个设置属性从 `periodic` 改为 `batch`，指定 Cassandra 必须在响应写操作之前将数据刷写到硬盘上。改变这个值需要进行一些性能考量，因为这做出了一些取舍：强制 Cassandra 更迅速地写入限制了它更自由地管理资源。如果将 `commitlog_sync` 设置为 `batch`，就必须给 `CommitLogSyncBatchWindowInMS` 一个合理的值，这里 MS 是每次同步写入的毫秒数。而且，在多节点集群中，通常没有必要这么设置，因为根据多副本定义，在其他节点也得到数据之前是不会响应写请求的。

如果你决定使用 `batch` 模式，需要把 commit log 分到独立的设备上来降低性能影响。不论何时，将 commit log 和 SSTable（数据文件）放到不同的磁盘上都是个好主意，即使你没有选择 `batch` 模式。

11.4 memtable

每个列族都有一个单独的 memtable 与之相关联，有一些设置是关于 memtable 处理的。在刷写到硬盘、成为 SSTable 之前，memtable 可以长大到的尺寸是由 `MemtableSizeInMB` 元素（在 YAML 中是 `binary_memtable_throughput_in_mb`）设置的。注意，这个值是 memtable 本身在内存中的大小，而不是占用的堆大小，因为一些列索引相关的开销，后者可能会更大。

你需要平衡设置这个参数和 `MemtableObjectCountInMillions`，这是 memtable 被刷写到硬盘之前，在其中可以存储的列值数量的阈值。

一个相关设置是 `memtable_throughput_in_mb`。这是一个 memtable 在被刷写到磁盘、成为 SSTable 之前可以存储的最大列数。默认值为 0.3，大约是 333 000 列。

你还可以配置 memtable 在刷写到硬盘之后，可以在内存中保存多久。这个值通过 `memtable_flush_after_mins` 元素设置。当刷写执行时，它会写入一个刷写缓冲区，也可以使用 `flush_data_buffer_size_in_mb` 配置这个缓冲区的大小。

另一个用于调整 memtable 的相关元素是 `memtable_flush_writers`。这个设置的默认值是 1，它是当 memtable 写入磁盘时需要使用的线程数量。如果有非常大的堆，把这个值调高可以提高性能，因为这些线程在磁盘 I/O 时会阻塞住。

11.5 并发

Cassandra 和很多数据存储系统的一大不同，就是提供了比读性能更高的写性能。有两个设置是关于多少个线程可以执行读操作和写操作的：`concurrent_reads` 和 `concurrent_writes`。一般来说，Cassandra 默认给出的设置就非常不错。但你可能还是需要在启动服务器之前修改 `concurrent_reads` 设置，因为 `concurrent_reads` 设置在每个处理器核两个线程的时候是最优的。默认情况下，这个设置是 8，假设服务器是 4 核的。如果这正是你的配置，那你刚好可以使用它。但如果你有一个 8 核的服务器，就应该把它设置为 16。

`concurrent_writes` 设置的形式有些不同。它应该设为将会并发访问到服务器的客户端的数量。如果将 Cassandra 作为一个 Web 应用服务器的后端，则可以把这个值从默认的 32 调到应用服务器可以用于连接 Cassandra 的线程数。对于 WebLogic 之类的 Java 应用服务器来说，通常合理的数据库连接池不会大于 20 或 30。不过，如果你使用了一个多应用服务器集群，可能还需要给这个值乘上一个因子。

11.6 缓存

有很多缓存相关的设置，既有 Cassandra 的设置，也有操作系统级的设置。缓存可能会占用相当大的内存，所以需要根据使用情况来小心地调整它们。

Cassandra 中有两类主要的缓存：行缓存和键值缓存。行缓存会缓存整行数据（它们所有的列），所以它是键缓存的超集。如果你对某个给定的列族使用了行缓存，那么你也就不需要对它使用键值缓存了。

因而，缓存策略应该根据下面几个条件确定。

- 考虑查询，使用更适合查询方式的缓存类型。
- 考虑堆内存和缓存尺寸的比例，不要让缓存占满堆容量。
- 考虑行尺寸和键值尺寸。通常键值都会比整行内容小很多。

`keys_cached` 设置的是存储在内存之中的键位置——不是键值——的数量。这个值可以指定一个小数（0 和 1 之间的一个数）或是一个整数。如果使用了小数，是指定了一个要缓存的键值的百分比；而如果使用了一个整数，则是指定将要缓存的键值位置的绝对数值。



`keys_cached` 是一个列族的设置，某些列族比其他的列族更常被访问，所以不同的列族可以有不同的键值位置缓存数量。

这个设置会消耗很多内存，但是如果你的位置已经不是热点了，可以考虑做个折中。

`disk_access_mode` 的目的是允许将文件映射到内存，这样操作系统可以缓存读操作，这样就可以降低 Cassandra 内部的缓存的负载。这听起来很好，不过在实践中，`disk_access_mode` 是最没用的设置之一，而且目前也无法按照预想的那样工作。这可能会在将来有所改进，不过看起来这个设置同样可能被移除。当然可以尝试它，不过你可能不会看到什么不同。

你还可以在服务器启动时就填入行缓存。要这么做，可以使用 `preload_row_cache` 元素。这个设置默认是 `false`，不过可以把它设置为 `true` 来改进性能。代价是，如果要预装载很多列族的数据，自举的时间可能会更长。

`rows_cached` 设置指定了将被缓存的行数。默认情况下，这个值是 0，意味着不使用行缓存，使用行缓存是一个好主意。这里，如果使用一个小数，那么就是指定全部数据要缓存的百分比；而如果是个整数，则就是要缓存位置的行数的绝对数值。不过，你应该谨慎使用这个设置，因为这个设置很容易失去控制。如果列族的读请

求大于写请求，那么把它设置得很高将会不必要地消耗太多的服务器资源。而如果列族有一个很低的读写比例，但有很大的行（包含上百列），那么在把这个值设得很高之前需要好好计算一下。除非某些行会有很高的命中率，而其他行不太会被访问到，否则你不会从这里得到很大性能提升。

11.7 缓冲区尺寸

缓冲区尺寸代表执行某些操作时分配的内存大小。下面是这些设置的一个概览：

- `flush_data_buffer_size_in_mb`
  这个值默认设为 32MB，这是用于将 memtable 刷写到磁盘上的缓冲区尺寸。

- `flush_index_buffer_size_in_mb`
  这个值默认设为 8MB。如果每个键值只定义了很少的列，可以考虑增加索引缓存的尺寸。相反，如果每行都有很多列，那么就应该减小这个缓冲区的尺寸。

- `sliced_buffer_size_in_kb`
  依赖于查询的变化程度有多高，这个设置可能没有太大用处。它允许你制定一个 KB 级的值，指定当执行切片查询时对邻近列进行的缓存。如果某个切片查询的执行概率远高于其他查询，或者数据分步中列族的列数量比较具有一致性，那么这个设置在读查询时可能会有一些用。不过记住，这个设置是全局的。

11.8 使用Python压力测试

Cassandra 附带了一个称为 `py_stress` 的受欢迎的工具，可以用于对 Cassandra 进行压力测试。要运行 `py_stress`，可以进入 <cassandra-home>/contrib 目录。你需要先阅读 README.txt 文件，这里面列出了要运行这个工具需要满足的先决条件。

在运行这个工具之前还有几个工作要做。首先确定配置中还有默认 keyspace（`Keyspace1`）并加载了它，因为这个工具会执行在默认的列族定义之上。

之后，你需要生成 Python Thrift 接口，这可能还需要几步操作。

11.8.1 生成Python Thrift接口

在运行压力工具之前，我们需要确定已经有 Python 的 Thrift 接口可用了（因为这是一个 Python 脚本）。如果执行命令时出现下面的一行错误，就说明还没有生成接口，或是生成的不正确：

```
No module named thrift.transport
```

还要确定已经在系统中安装 Python 了。要验证是否安装了 Python，打开终端窗口，键入 **$python**，你应该可以看到类似下面的输出：

```
eben@morpheus$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information
```

要确保 Python 版本是 2.6 或者更高，这样你才能得到它们提供的最新的多线程能力，这对于压力测试来说非常方便。

获得Thrift

要获得 Thrift，从 http://incubator.apache.org/thrift 下载，你将得到一个类似 thrift-0.2.0-incubating.tar.gz 的文件。将它解压，然后运行如下 Linux 命令来建立依赖关系[1]：

```
$ sudo apt-get install -y libboost-dev libevent-dev python-dev automake
    pkg-config
libtool flex bison
```

如果系统中没有安装好 C++ Boost 库，Thrift 可能无法正常工作。可以从 http://www.boost.org 下载 Boost。之后，在 Boost 目录中，运行如下命令：

```
$ ./bootstrap.sh
$ ./bjam
```

这将编译和安装 Boost。现在，要编译 Thrift，可以以 root 权限在 Thrift 目录中运行如下一组命令：

```
$ ./bootstrap.sh
$ ./configure
$ ./make
$ ./make install
$ cd lib/py
$ ./make
```

现在运行命令 `$which thrift`，就会告诉你 Thrift 安装到的路径了。在我的系统里是 /usr/local/bin/thrift。

译注 1：这个命令仅适用于使用 APT 包管理系统的 Linux 发布版，如 Debian 和 Ubuntu 等，其他发布版需要使用相应的包管理工具。



在 Ubuntu Linux 中，在安装 Thrift 时可以看到这样一个错误“autoscan: not found”。yum 或者 apt-get 中都没有一个名为 autoscan 的包。要获得 Thrift 自举需要的 autoscan，可以运行如下命令：

```
$ sudo apt-get install automake
```

可以通过 **$which autoscan** 来查看是否安装了 autoscan 了，如果返回值是空，就是还没有安装它。

如果 Thrift 已正确安装了，应该可以运行 Thrift 程序并看到帮助输出：

```
$ thrift -version
Thrift version 0.2.0-exported
```

把 site-package 目录放在 Python 路径中：

```
$ export PYTHONPATH=/usr/lib/python2.6/site-packages/
```

现在可以进入 <cassandra-home> 目录了。运行下面的命令来为 Python 生成 Thrift 接口：

```
$ ant gen-thrift-py
```

你应该可以得到一个编译成功的消息。现在，<cassandra-home>/interfaces/thrift/gen-py 目录下就有了 Cassandra 的 Thrift 为 Python 生成的接口。将这个目录复制到 <cassandra-home>/contrib/py_stress。现在只要确定 Cassandra 服务器已经启动，就可以运行 Python 压力测试了。

11.8.2 运行Python压力测试

现在一切设置妥当，可以运行压力测试了。进入 <cassandra-home>/contrib/py_stress 目录。在终端中，输入 **stress.py** 对本机运行测试。

如果看到的消息表示压力测试无法连接 `localhost:9160`，那你有几种选择。首先，确定 Cassandra 确实启动了，并且在监听这个地址和端口。如果配置 Cassandra 为监听 IP 或主机名，而非 localhost，就需要改动脚本，让它指向服务器，或者修改服务器的配置来监听 localhost，或者可能是最好的方法，通过传送 `-d` 参数来告诉脚本连接哪个服务器。让我们开始吧：

```
$ stress.py -d 192.168.1.5
```



你可以运行 `stress.py -h` 来查看压力测试脚本的使用方法。

这个测试将会插入一百万个值，然后停止。在一个装备了 Intel i7 处理器（类似 8 核

的处理器）、4 GB 内存，同时运行了很多其他进程的普通工作站上，我运行了这个测试。这里是我的输出：

```
eben@morpheus$ ./stress.py -d 192.168.1.5 -o insert
total,interval_op_rate,avg_latency,elapsed_time
196499,19649,0.0024959407711,10
370589,17409,0.00282591440216,20
510076,13948,0.00295883878841,30
640813,13073,0.00438663874102,40
798070,15725,0.00312562838215,50
950489,15241,0.002910990841 7,60
1000000,4951,0.00444872583334,70
```

我来解释一下。我们做的事情是将一百万个值插入到一个未经调优的 Cassandra 服务器上，用时大约 70 秒。可以看到，前 10 秒插入了 196 699 个随机生成的值。平均操作时延时 0.0025 秒，即 2.5 毫秒。但这是使用默认配置的，并且在启动测试之前的 Cassandra 服务器中，已经有了大约 1GB 的数据。我们提供更多的线程，来看一下是不是能得到一些性能改进：

```
eben@morpheus$ ./stress.py -d 192.168.1.5 -o insert -t 10
total,interval_op_rate,avg_latency,elapsed_time
219217,21921,0.000410911544945,10
427199,20798,0.000430060066223,20
629062,20186,0.000443717396772,30
832964,20390,0.000437958271074,40
1000000,16703,0.000463042383339,50
```

这里，使用了 -t 参数，指定一次使用 10 个线程。这里显示，在 50 秒钟内插入了一百万条记录，大约每条写操作 2 毫秒时延。这是使用的一个完全未经调优并已经包含大约 1.5 GB 数据的数据库。

你应该多次运行测试来得到根据硬件设置最合适的线程数。根据处理器核数，如果使用过高的线程数，将会得到更差而不是更好的性能，因为处理器会花费更多的时间用于管理线程，而不是处理工作。你应该让线程数和处理器核数基本匹配来得到一个合理的测试结果。

现在已经把数据放入到数据库中了，让我们使用测试工具来读取一些值：

```
$ ./stress.py -d 192.168.1.5 -o read
total,interval_op_rate,avg_latency,elapsed_time
103960,10396,0.00478858081549,10
225999,12203,0.00406984714627,20
355129,12913,0.00384438665076,30
485728,13059,0.00379976526221,40
617036,13130,0.00378045491559,50
749154,13211,0.00375620621777,60
880605,13145,0.00377542658007,70
1000000,11939,0.00374060139004,80
```

可以看到，Cassandra 的读速度不像写速度那么快，花费了大约 80 秒才读出一百万条记录。记住，这是未经调优、默认设置在单线程上，而且运行了其他程序的普通工作站上运行的测试，而且数据库里已经存有 2 GB 的数据。不过，这是一个不错的工具，帮你针对自己的环境进行性能调优而且能大致了解集群性能。

11.9 启动和JVM设置

Cassandra 允许你配置很多关于如何启动、Java 需要分配多少内存之类的选项。在本节中，我们来看如何调优启动参数。

如果使用 Windows，则启动脚本是 cassandra.bat；而如果使用 Linux，则应该是 cassandra.sh。可以直接运行这个文件来启动服务器，这里面设置了很多默认值。不过，在 bin 目录里还有另一个文件允许我们配置各种关于 Cassandra 启动的设置。这个文件称为 cassandra.in.sh，里面包含了很多选项，包括 JVM 设置等，把设置放到一个单独的文件中可以让升级维护更加方便。

JVM调优

为了得到更好的性能，可以对传给 JVM 的启动参数进行调优。关键的 JVM 选项已经包含在 cassandra.in.sh 文件中了，对它们进行调优的指南如表 11-1 所示。如果要修改这些值，只要在文本编辑器中打开并修改这些值，然后重新启动 Cassandra 就可以了。

表11–1　Java性能调优选项

Java选项	设置指南
最大与最小堆尺寸	这两个设置分别默认为 256 MB 和 1 GB。要调优这个设置，可以把它们设置得更高，并设置为一样的值（参考后面的提示）
断言（assertion）	默认情况下,JVM 使用 `-ea` 开关打开了断言。把 `-ea` 改为 `-da`(关闭断言）可以提升性能
存活比例（survivor ratio）	Java 对分成了两块对象区域：年轻的和年老的。年轻空间中，一部分用于新对象分配（称为“伊甸园”（eden space）），一部分用于存放还在使用的新对象。年老对象是那些经历过几次垃圾回收仍然被引用的对象。存活比例是年轻代的堆空间里，eden space 对 survivor space 的比例。如果应用中会创建很多新对象，但生存周期较短，就可以把这个值设高一些。反之，如果应用中大部分都是长期存活的对象，就应该把这个值设低一些。Cassandra 这个设置的默认值为 8，也就是说，eden 与 survivor 的空间比例是 1:8（每个 survivor 空间的尺寸将是 eden 的 1/8）。这个比例相当低，因为 memtable 里的对象的生存时间都很长。这个设置可以和 `MaxTenuringThreshold` 一起调整
`MaxTenuringThreshold`	每个 Java 对象的标题处都有一个年龄域，表示它在年轻代里被复制的次数。它们每经历一次年轻代的垃圾回收就会复制一次，而这个复制是有开销的。因为长期存在的对象可能会被复制很多次，调试这个值可以改进性能。默认情况下，Cassandra 设置这个值为 1。可以把它设置为 0，这样会立刻把存活过一次的年轻代 GC 的对象放入年老代

（续）

Java选项	设置指南
UseConcMarkSweepGC	这个选项指示JVM的垃圾收集（garbage collection, GC）策略，特别地，它启动了并发标记算法。这个设置会占用更多的RAM和更多的CPU的资源，在程序运行的时候进行更频繁的GC，以保证GC造成的中断时间最小。使用这个策略，应该将堆的最小值和最大值设置为相同值，这样防止JVM花费很多时间来增长堆的尺寸。也可以使用-XX:+UseParallelGC，这可以利用多处理器机器的资源。这样能有更好的峰值性能，但会有偶尔的停顿。不要对Cassandra使用Serial GC

在 include 主配置文件中的主要参数包装了 Java 的设置。比如，默认设置最大 Java 堆内存为 1 GB。如果你有一个拥有更多内存的机器，可以调整这个设置。可以设置相同的 -Xmx 和 -Xms 值来防止 Java 管理堆增长带来的开销。



32 位 JVM 堆尺寸的理论最大值是 4 GB。然而，不要简单地将 JVM 设置到可用的全部 4 GB 内存。这里包含了很多因素，比如交换空间和内存碎片。如果没有可用的交换空间，简单使用 -Xmx 来增加堆尺寸不会获得性能收益。一般情况下，在 32 位 Windows 下的 JVM 可以获得大约 1.6 GB 的可用堆空间，而在 Solaris 下可以接近 2 GB。在 64 位系统中使用 64 位 JVM 可以得到更多的空间。可以从 http://java.sun.com/docs/hotspot/HotSpotFAQ.html 获得更多信息。

调试这些设置可以让压力测试程序表现更好。比如，我的机器使用如下设置可以比默认设置提高 15% 的性能：

```
JVM_OPTS=" \
        -da \
        -Xms1024M \
        -Xmx1024M \
        -XX:+UseParallelGC \
        -XX:+CMSParallelRemarkEnabled \
        -XX:SurvivorRatio=4 \
        -XX:MaxTenuringThreshold=0
```

在进行性能调优的时候，一个好的方法是先只设置最大和最小堆空间，而不设置其他值。只有在实际使用环境下或某些性能评分系统中，使用堆分析工具，并观察特定应用行为的情况下，才能进入高级 JVM 设置。如果你调整了 JVM 参数，看到在使用负载测试工具或其他类似 contrib 中的 Python 性能测试工具时取得了好成绩，不要过于兴奋，你需要在真实世界中测试，不要简单复制这些设置。



对于 Java 6 性能调优（Java 6 和之前版本的操作有些不同），可以参考 http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html[2]

译注 2：译者曾将此文译为中文：http://wangxu.me/blog/p/209，供参考。

通常，如果你遇到无可用内存的错误，可能需要进行堆来转储它的状态。如果内存错误问题解除，这是一个很好的方法。你还可以通过打印垃圾回收的细节来了解问题。此外，如果 Cassandra 中有很多数据，而且垃圾回收导致了很长的停顿，可以尝试在堆中内存不太满时就开始运行垃圾回收。所有这些参数如下所示：

```
-XX:CMSInitiatingOccupancyFraction=88 \
-XX:+HeapDumpOnOutOfMemoryError \
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -verbose:gc \
```

11.10 小结

在本章中，我们看了 Cassandra 中可用的有助性能调优的设置，包括缓存设置、内存设置和硬件考量。我们还设置并使用了 Python 压力测试工具，写入然后读出了一百万行数据。

如果你对 Linux 系统不太熟悉，又希望在 Linux 上运行 Cassandra（这是推荐的方式），可以参考 Jonathan Ellis 的博客文章，其中介绍了很多 Linux 性能监控工具，可以帮你了解底层平台的性能，这样你可以在合适的地方解决问题。这篇文章可以在 http://spyced.blogspot.com/2010/01/linux-performance-basics.html 找到。

终极的性能提升手段就是在能负担得起的范围内，配置很多内存以及尽量多的 CPU 核。

第 12 章

集成 Hadoop

Jeremy Hanna

随着越来越多的公司和组织开始采纳 Cassandra 这类技术，他们也开始寻找用于对数据进行分析与查询的工具。Cassandra 内建的查询方式配合着其上的应用层，提供了相当丰富的查询手段。不过在软件业中有同样适合和 Cassandra 一起工作的分步式分析工具。

Hadoop 看起来就是开源海量数据处理框架的不二之选。其中有开源的 MapReduce 实现，也有架构其上的高级查询引擎，如 Pig 和 Hive。感谢 Cassandra 和 Hadoop 团队成员们的努力，目前 Cassandra 已经在与 Hadoop 及其分析工具的集成方面有了显著的进展。

在本章中我们将探讨如何让 Cassandra 和 Hadoop 共同工作。首先，我们简述一下 Apache Hadoop 项目的简史，然后介绍如何为存储在 Cassandra 之中的数据写 MapReduce 程序。之后，我们将探讨与 Hadoop 之上的高级查询工具的集成：Pig 和 Hive。了解了这些工具之后，我们还会介绍如何配置 Cassandra 集群，让这些分析可以分布式进行。最后，我们会分享一些使用 Cassandra 和 Hadoop 来共同解决现实世界问题的案例。

12.1　何为Hadoop

如果你已经很熟悉 Hadoop 了，完全可以跳过本节。Hadoop（http://hadoop.apache.org）是一组用于分布式处理大量数据的开源项目。其中的 Hadoop 分布式文件系统

（HDFS）和 MapReduce 子项目是 Google GFS 和 MapReduce 的开源实现。

Google 发现，很多内部项目组都在实现类似的功能，用以分布式解决问题。他们看到，很多分布式数据处理操作都可以分解为两个阶段：map 阶段和 reduce 阶段。map 函数对原始数据进行操作，生成中间值。reduce 对这些中间值进行某种处理，生成最终的 MapReduce 计算结果。通过对公共框架进行标准化，他们可以创建更多的问题解决方案，而不是做换汤不换药的无用功。

Doug Cutting 决定写一个开源应用，基于 Google 文件系统（http://labs.google.com/papers/gfs.html）和 MapReduce（http://labs.google.com/papers/mapreduce.html），Hadoop 就此诞生。以此为起点，Hadoop 迅速发展，成为了一系列处理海量数据问题的工具。如今，Hadoop 得到了非常广泛的应用，使用者包括 Yahoo!、Facebook、LinkedIn、Twitter、IBM、Rackspace 以及很多其他公司。这是一个充满生机的团队和一个成长中的生态系统。

Cassandra 内建了对 Hadoop MapReduce 的支持（http: //hadoop.apache.org/mapreduce）。

12.2 使用MapReduce

本节详述如何使用 Java 语言对存储在 Cassandra 中的数据写一个简单的 MapReduce 程序。我们还会简单介绍如何将输出数据写入到 Cassandra，并讨论正在开发的工作：使用 Cassandra 和 Hadoop Streaming 来支持 Java 之外的语言。



本节中给出的字数统计（word count）例子可以在 Cassandra 源代码的 contrib 部分找到。这个例子可以按照本节的介绍编译运行。最好使用源代码包中的版本来运行，因为当前的版本相对于本书可能又有一些改进了。不过，基本原理是一样的。

为方便起见，字数统计 MapReduce 程序会运行在一个 Cassandra 本地节点。关于如何配置 Cassandra 和 Hadoop，更分布式地运行 MapReduce，可以参考 12.5 节。

Cassandra Hadoop源码包

Cassandra 有一个用于与 Hadoop 集成的 Java 源码包，称为 `org.apache.cassandra.hadoop`，其中包括如下内容。

- `ColumnFamilyInputFormat`

  我们将主要用这个类来让 Hadoop 从 Cassandra 交互读取数据。它扩展了 Hadoop 的 `InputFormat` 抽象类。

- `ConfigHelper`

  一个用于配置 Cassandra 相关信息的辅助类，比如服务器节点位置、端口以及特定 MapReduce 任务相关的信息。

- `ColumnFamilySplit`

  这个类扩展了 Hadoop 的 `InputSplit` 抽象类，对 Cassandra 数据进行分解。它还向 Hadoop 提供数据的位置信息，这样可以把任务调度到存储数据的节点上。

- `ColumnFamilyRecordReader`

  从 Cassandra 读取单条记录的层。它扩展了 Hadoop 的 `RecordReader` 抽象类。

此外，在 Hadoop 包里，还有一些将数据输出到 Cassandra 的相似的类，但是在本书写作的时候，这些类还没有最终完成。

12.3 运行字数统计例子

字数统计是在 MapReduce 论文中给出的一个例子，是很多初学者学习这个框架的起点。它读入一个文本正文，统计每个不同的词的出现次数。这里我们给出一些代码，来对存储在 Cassandra 里的数据进行字数统计。在 Cassandra 的源码包里也包含一份可以运行的字数统计的例子。

首先，我们需要一个 `Mapper` 类，如例 12-1 所示。

例 12-1：TokenizerMapper.java 类

```
public static class TokenizerMapper extends Mapper<byte[],
    SortedMap<byte[], IColumn>, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  private String columnName;

  public void map(byte[] key, SortedMap<byte[], IColumn> columns, Context
      context)
    throws IOException, InterruptedException {

    IColumn column = columns.get(columnName.getBytes());
    String value = new String(column.value());
    StringTokenizer itr = new StringTokenizer(value);

    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
  }
  protected void setup(Context context)
    throws IOException, InterruptedException {
```

```
        this.columnName = context.getConfiguration().get("column_name");
    }
  }
```

熟悉 MapReduce 程序的读者会发现，这个 mapper 看起来十分眼熟。在这里，mapper 的输入是从 Cassandra 读取的行键值和相关的行值。行值在 Cassandra 世界里直接对应为包含列信息的映射。此外，对于字数统计代码来说，我们覆盖了 setup 方法来设置我们要找的列名字。其余的 mapper 代码就是普通的 word count 实现。



当 mapper 迭代超级列时，每个 `IColumn` 都需要类型转换为 `SuperColumn`，其中会包含内嵌的列信息。

接下来我们看一下字数统计程序的 `Reducer` 的实现，如例 12-2 所示。

例 12-2：Reducer 的实现

```
public static class IntSumReducer extends
  Reducer<Text, IntWritable, Text, IntWritable> {

  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable> values, Context
context)
    throws IOException, InterruptedException {

    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

在这个 reducer 实现中，没有什么奇怪的，没有 Cassandra 特有的东西。

最后看看运行 MapReduce 程序的类，如例 12-3 所示。

例 12-3：WordCount 类运行这个 MapReduce 程序

```
public class WordCount extends Configured implements Tool {

  public int run(String[] args) throws Exception {
    Job job = new Job(getConf(), "wordcount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setInputFormatClass(ColumnFamilyInputFormat.class);
    FileOutputFormat.setOutputPath(job, new Path("/tmp/word_count"));
```

```
        ConfigHelper.setThriftContact(job.getConfiguration(), "localhost", 9160);
        ConfigHelper.setInputColumnFamily(
            job.getConfiguration(), "Keyspace1", "Standard1");

        SlicePredicate predicate = new SlicePredicate().setColumn_names(
            Arrays.asList(columnName.getBytes()));

        ConfigHelper.setInputSlicePredicate(job.getConfiguration(), predicate);
        job.waitForCompletion(true);

        return 0;
    }
}
```

关于这个和一般样板类字数统计不完全相同的 `WordCount` 类，有些事情还是值得一提的。我们需要设置 `InputFormat` 为 Cassandra 的 `ColumnFamilyInputFormat`，以及制定 mapper 要查找的列名。Cassandra 包含了一个称为 `ConfigHelper` 的辅助类，提供了设置所需属性的方法，比如 Thrift 连接信息（服务器和端口）、keyspace 和列族等。它还允许我们设置切片谓词。

12.3.1　将数据输出到Cassandra

在这个例子里，reducer 使用了 `FileOutputFormat` 输出它的结果。像它的名字那样，`FileOutputFormat` 将结果写到文件系统中去。在 0.7 版本中，将有一个基于 Cassandra 的 `OutputFormat` 实现。在本章写作的时候，一些实现细节还没有最终完成。对于内建的输出格式的更新信息可以参考 http://wiki.apache.org/cassandra/HadoopSupport。

不过，也有可能直接在 `Reducer` 里通过 Thrift（或高级客户端）直接将数据写入到 Cassandra 中。在这个例子里，这意味着不写上下文，而是直接把键和值写到 Cassandra 中。

12.3.2　Hadoop流

字数统计 MapReduce 例子是用 Java 写成的。Hadoop 流（streaming）是 Hadoop 提供的允许 Java 之外的其他语言使用标准输入和标准输出进行 MapReduce 工作的方法。在本章写作时，Hadoop 流还不支持 Cassandra 的 MapReduce 集成。不过这个支持将在不远的将来实现。对于当前的状态，可以参考 wiki。

12.4　MapReduce之上的工具

MapReduce 是为开发者提供的一个伟大的抽象，这样他们可以更少地考虑分布式计算细节，更多地考虑他们要解决的问题了。随着时间的推移，更抽象的工具集也出

现了。Pig 和 Hive 运行在 MapReduce 之上，并允许开发者更容易地进行更复杂的分析。两者都可以运行在 Cassandra 的数据之上。

12.4.1 Pig

Pig（http://hadoop.apache.org/pig）是一个 Yahoo! 开发的数据分析平台。这个平台包含了一个名为 Pig Latin 的高级语言和一个编译器，会将使用 Pig Latin 写成的程序编译为在 MapReduce 作业的序列。

开发者们在集成 Hadoop，使用 Cassandra 中的数据运行 MapReduce 工作的同时，还进行了一些 Pig 集成的工作。通过使用 Pig 的 grunt shell 提示符和 Pig Latin 脚本语言，Pig 提供了一种简化查询代码编写的方法。使用 Pig Latin 来运行字数统计例子，只要这样：

```
LOAD 'cassandra://Keyspace1/Standard1' USING CassandraStorage() \
as (key:chararray, cols:bag{col:tuple(name:bytearray, value:bytearray)});
cols = FOREACH rows GENERATE flatten(cols) as (name, value);
words = FOREACH cols GENERATE flatten(TOKENIZE((chararray) value)) as word;
grouped = GROUP words BY word;
counts = FOREACH grouped GENERATE group, COUNT(words) as count;
ordered = ORDER counts BY count DESC;
topten = LIMIT ordered 10;
dump topten;
```

这个字数统计只有 8 行长。第一行读取了所有 `Standard1` 列族的数据，描述了数据的别名和数据类型。我们从每行中提取了名 / 值对。在第三行中，我们必须将值转换成一个字符数组的类型，让它可以用于内建的 `TOKENIZE` 函数。然后对每个词进行分组和技术。最后，对数据按照计数进行排序，并输出找到的前十位的词。



使用 Pig 处理超级列是很平常的操作，只是有一层嵌套数据而已，我们可以将它扁平化，从而获取里面的值。

对于有些人，编写 MapReduce 程序非常乏味而且充满了样板代码。Pig 提供了一层抽象，让代码简洁了很多。相比于只使用 MapReduce，Pig 还允许程序员更简单地表达 join 之类的操作。

Pig 集成代码（一个 `LoadFunc` 的实现）位于 Cassandra 源码的 contrib 部分。它可以按照目录中提供的介绍，编译并运行。它还包含了一些如何配置 Cassandra 特有配置选项的介绍。现在，我们可以看看如何配置一个 Cassandra 集群来分布式地运行 Pig 任务了（会编译为 MapReduce）。

12.4.2 Hive

和 Pig 类似，Hive（http://hadoop.apache.org/hive）是一个数据分析平台。Hive 并不使用脚本语言，而是使用一种类似于 SQL 的称为 Hive-QL 的查询语言进行查询。Hive 由 Facebook 开发，允许把大数据集抽象为通用的结构。

在本章写作时，Hive 的 Cassandra 存储处理工作即将完成。对于 Hive 与 Cassandra 的共同使用的更新和文档，可以参考 wiki。

12.5 集群配置

MapReduce 和其他工具在尝试事物或解决问题的时候，可以非分布式地运行。但是要运行在生产环境中，需要把 Hadoop 安装到 Cassandra 集群上来。虽然深入讨论 Hadoop 的安装和配置超出了本章的范围，但我们会简单介绍如何让 Cassandra 和 Hadoop 配置在一起以获得最佳性能。读者们可以从 http://hadoop.apache.org 找到更多关于 Hadoop 配置的信息，或者阅读 Tom White 的杰作《Hadoop 权威指南》（*Hadoop: The Definitive Guide*）（O'Reilly）[1]。

因为 Hadoop 有一些大家不熟悉的专有名词，我们这里给出一些有用的定义。

- `HDFS`
  Hadoop 分布式文件系统。

- `namenode`
  HDFS 的主节点。它拥有存储于多个 Datanode 的数据块的位置信息，在小集群里，经常和 jobtracker 运行在同一台机器上。

- `datanode`
  为 HDFS 存储数据块的节点，datanode 和 tasktracker 运行在相同的节点上。

- `jobtracker`
  MapReduce 工作的主节点。jobtracker 接收新工作，将工作分为 map 和 reduce 任务，并将任务交给集群中的 tasktracker 完成。它负责工作的完成。在小集群中，它经常和 namenode 运行在同一个节点上。

- `tasktracker`
  负责为 jobtracker 运行 map 或 reduce 任务的进程。tasktracker 和 datanode 运行在相同的服务器上。

译注 1：也可参考即将由人民邮电出版社出版的《Hadoop 实战》。

和 Cassandra 一样，Hadoop 也是一个分布式系统。MapReduce jobtracker 将任务分配到整个集群上，尽量贴近数据所在的位置。当 jobtracker 启动一个任务时，它从 HDFS 得到数据存储位置的信息。类似地，Cassandra 内建的 Hadoop 集成也会向 jobtracker 提供数据位置信息，这样任务就可以更接近数据了。

为了达到数据本地化，Cassandra 节点也必须是 Hadoop 集群的一部分。namenode 和 jobtracker 可以在 Cassandra 集群之外的服务器上。Cassandra 节点需要成为集群的一部分而在每个节点上都运行 tasktracker 进程。接下来，当一个 MapReduce 任务启动后，jobtracker 在分配 map 和 reduce 任务时，可以向 Cassandra 查询数据的位置信息。

图 12-1 示意了一个四节点的 Cassandra 集群，每个 Cassandra 节点上都运行着 tasktracker 进程。集群中至少有一个节点需要运行 datanode 进程。少量数据（分布式缓存）仍然需要 HDFS，而一个单独的 datanode 应该足够了。集群外的服务器运行着 Hadoop 的 namenode 和 jobtracker。

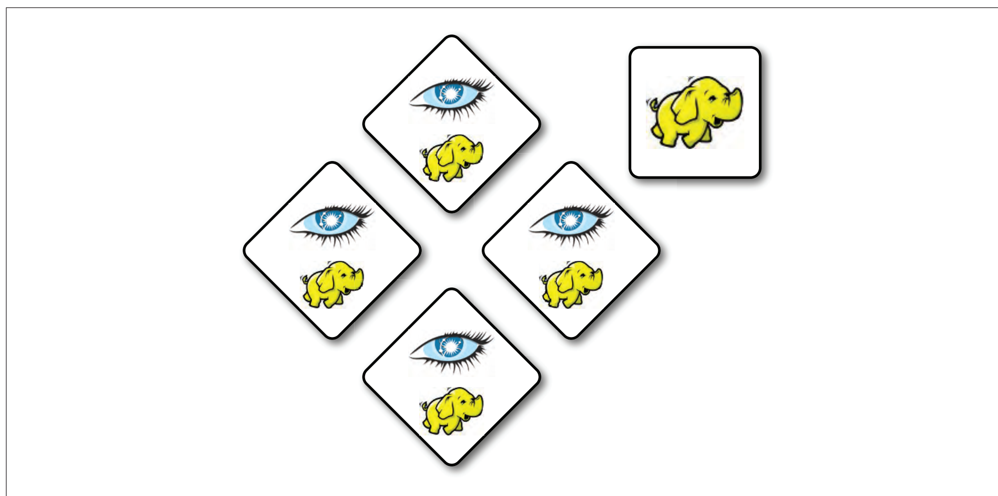


图 12-1：四节点的 Cassandra 集群，外部有 namenode/jobtracker

当一个客户机发起任务时，它会到达 jobtracker。jobtracker 在工作提交时，通过开始提到的配置选项得到数据源信息。这时它可以使用 Cassandra 的 `ColumnFamily-RecordReader` 和 `ColumnFamilySplit` 来获取不同片段的数据在集群中的物理位置。之后，它可以使用位置信息将任务分配到集群中的节点上，尽量让任务运行在有相关数据的节点上。

最后，当创建工作执行 MapReduce 的时候（直接或是通过 Pig），Hadoop 的配置需

要（根据 Hadoop 配置文件）知道 namenode/jobtracker，以及 Cassandra 的配置选项。这样，集群就可以支持 Hadoop 与 Cassandra 的集成了。

12.6 案例

为了帮助你理解为什么 Cassandra/Hadoop 的集成是有趣又有用的，我们将会介绍一两个 Cassandra 使用者们的案例。

12.6.1 Raptr.com: Keith Thornhill

Raptr 是一个允许游戏玩家相互通信并共享游戏统计和成就的服务。Keith Thornhill 是 Raptr 的高级软件工程师，他发现需要让他们的后台存储和分析更上一个台阶。他们老的存储方案和分析方法是最早的土生土长的方法，已经不适应现在的发展了。在整个数据集中进行查询非常困难，可能需要运行几个小时。

Keigh 认为 Cassandra 是一个很有前途的存储方案，因为它：

- 内建的可扩展性，而非建在鹰架上的；
- 对于读写访问呈现单一视图（没有主从之分）；
- 在正常情况下非常易于维护（包括节点失败、增加新节点等），“直接可用”，只需要很少的微管理。

Keith 也在关注 Cassandra/Hadoop 的集成工作的演进，并把 Pig 看做是一个他可以使用的分析解决方案。起初他希望找到通过 PHP 或是 Python 使用 MapReduce 的方法。但是，当他熟悉了 Pig 之后，就没有这样的需求了。他指出，从有想法到执行 Pig 任务的周转时间可以非常少。查询时间也让人惊喜。他可以在 10~15 分钟内遍历所有数据，而非几个小时。作为结果，Raptr 已经有能力寻找新的分析数据的可能性了。

对于配置，Keith 使用了独立的 namenode 和 jobtracker，并在每个 Cassandra 节点上安装了 datanode/ tasktracker。他认为，这样的一个很好的副作用就是，分析引擎也可以和数据一起扩展。

12.6.2 Imagini: Dave Gardner

Imagini 为发行商提供工具，通过“可视实验”和推理引擎对它们站点的访问者进行分析。在幕后，这需要处理大量的行为数据，然后让数据可以实时访问。

在调研了很多替代方案之后，Imagini 因为 Cassandra 的容错性、无中心架构（没有单点失效）和强大的写能力而选择了它。

Dave Gardner 是 Imagini 的一位高级开发者，他写道："我们使用 Cassandra 存放实时数据，包括大约 1 亿用户的信息，而且在未来一年中还会快速增长。这些数据差不多都是简单的键值查找。"

目前，Imagini 从各种数据源收集数据放入 Hadoop 分布式文件系统（HDFS）之中。使用 Hadoop 流，他们使用 PHP 对数据进行 MapReduce 处理，并在 reducer 里，通过 Thrift 将输出直接送入 Cassandra 里。在 Cassandra 中的结果可以提供实时的数据访问。

一旦 Cassandra 支持了 Hadoop 流，Imagini 就有希望进一步简化数据流了。他们甚至计划把原始数据直接存放在 Cassandra 之中，在数据上进行 MapReduce，然后直接输出到 Cassandra。

12.7 小结

本章中，我们学习了 Hadoop——Google MapReduce 算法的开源实现。我们了解了 Hadoop 的基本概念，然后通过一个字数统计的例子学习了如何将它与 Cassandra 集成在一起，还看到了如何使用 Pig，一种简单而简洁的用于表达 MapReduce 工作的语言。

最后，我们了解了一些公司在集成使用 Hadoop 和 Cassandra 时的状况。

附录

非关系型数据库大观

Cassandra 只是冉冉升起的众多新的非关系型数据库项目中的一个，为了了解非关系型数据库的设计目标，以及基于这些目标而采用的种种具体设计，我们来探究一下这些不同的项目。



近来，我们时常听到“NoSQL”这个名词，用来描述一些不使用 SQL 的数据库。我也一直使用“非关系型”这个名词来总称这些数据库。不过，本附录中，我们会解释为什么不应该从这个角度来描述这些产品。从这个角度对比 NoSQL 数据库和传统关系型数据库，没有实际意义，接下来你就会看到，所有这些所谓的“NoSQL”数据库本身各不相同，不论是具体实现，还是目标、特性、优缺点等，都各有千秋。所以，比较“NoSQL”和“关系型”是个不折不扣的骗局。

在本附录中，我们将审视多种流行的非关系型数据库。Cassandra 非常擅长做某些事情，但做其他事情不见得在行。所以我的目标是帮助你理解 Cassandra 在多种非关系型数据库中所处的位置，以根据需求来做出正确的选择。即使你已经明确希望使用 Cassandra 了，这个调研仍然可以帮你理解 Cassandra 的很多设计决策和取舍之处。

A.1　非关系型数据库

毫无疑问，世界上随处可见那些没有使用半点关系模型的流行数据库产品。它们包括对象数据库、原生 XML 数据库、面向文档的数据库、图数据库以及键 - 值存储系统。其中的一些代表产品已经使用很多年了，另一些则刚刚开始产品应用。我在这里讨论几种非关系型数据库。



这里有很多非关系型数据库没有提到，主要是因为它们或者少有人问津，或者是专用系统，抑或是还没有什么具体实现或是产品级应用。其中之一就是“半关系型”数据库 Drizzle，这是一种基于 MySQL 的产品。微软基于 SQL Server 的云数据库平台称为 SQL Server 数据服务 (SDS)。还有 Yahoo! 的 PNUTS 毫无疑问也是值得一看的。PNUTS 的论文可以在这里找到：http://research.yahoo.com/files/pnuts.pdf。想要了解更完整的非关系型数据库，可以访问 Alex Popescu 的非常棒的网站 MyNoSQL：http://nosql.mypopescu.com。

你读这本书可能是因为已经为自己的项目选择了 Cassandra。当然，你也可能只是听说 Twitter 和 Facebook 这样的流行网站在使用 Cassandra，所以决定来进一步了解它。如果是后者，了解一些竞争产品，看看它们各自的专长、差异，以及 Cassandra 与它们相互间的对比，可能是非常必要的。

所以来看看这些替代产品，看看它们和我们已经熟悉的数据库产品有何异同。我会尽力使用同一套分类名词来描述各个产品的特性，以便进行比较。

总的来说，这些数据库都是分布式的，这意味着它们的设计允许用多个节点容纳数据的副本，自动管理副本。（不过也有例外，比如亚马逊的 SimpleDB。）它们拥有各种处理大规模扩展的特性，这对于很多新的网络应用非常重要。

不过，从另一个角度来看，这些数据库都还缺少优秀的工具和框架的支持。很多解决方案都很新，这意味着开发者们还集中在核心产品上，你不得不放弃那些图形化终端之类的工具，至少是暂时放弃，虽然这些工具在 RDBMS 中已经相当普遍了。要使用包括 Cassandra 在内的这些解决方案，需要适应命令行工作界面、简陋的 Shell 工具，以及做一些脏活累活。但是因为在这些数据库中，很多正在逐渐流行和获得认可，所以你可以期待不久的将来就有便利的工具可用了。

A.2 对象数据库

对象数据库的设计初衷是避免对象–关系模型之间的“阻抗不匹配”问题，在面向对象语言的项目里使用关系型数据库的时候常常面临这样的问题。对象数据库中的数据并不按照关系和行列来存储，而是直接以对象的方式来存储，这在面向对象应用中可以非常直接地使用。这可以让你在程序中回避多余的 SQL 代码，或是存储过程，以得到从应用对象到数据库表的映射，同时避免使用对象–关系映射（ORM）层。ORM 层可能会非常笨重，增加了应用代码的复杂性，并影响数据操作的效率。

因为在对象数据库里，你不需要再进行应用中的对象到关系型模型的数据转换，所

以应用可以运行得非常快。你还不需要使用关系键值进行 join 来应付查询，因为数据库里的数据和应用里一样，都可以通过指针来进行查找。

对象数据库从 20 世纪 70 年代中期到 80 年代就出现了，但从未获得广泛的商业应用，只在一些刚刚起步的边缘领域里获得成功，如计算机辅助设计（CAD）应用、空间应用、电信领域和嵌入式系统。

InterSystems 的对象数据库 Caché 可能是最为人所知的对象数据库商业产品了，此外，永久对象与可扩展数据库技术（POET，现在称为 Versant Object Database），也可以在 Java、.NET 和 C++ 应用中使用。

面向对象数据库有一些致命缺点。虽然使用面向对象数据库可以带来很多性能上的改善，但是也会让数据存储与应用过为紧密地耦合在一起，你需要进行取舍，看是否值得这么做。而且，对象数据库的数据序列化写入与读出通常只匹配同一种语言，这种更紧密的耦合关系同时也严重地限制了架构的灵活性。

近年来，相比于这里提到的其他存储系统，对象数据库较少受到关注，也鲜有发展，所以我也不会更详细地介绍它们了。

A.3 XML数据库

XML 数据库是文档数据库的一种特殊形式，专门为与 XML 一起应用而优化。XML 的第一个工作草案于 1996 年完成。1998 年 2 月，W3C 发布了 1.0 版的 XML 标准，很快 XML 就在全世界范围内得到了广泛的应用。众多互联网应用发现这种格式的语义非常丰富，很容易用作跨语言的传输介质。于是，所谓的“XML 原生”数据库很快就出现了，最早的之一就是 Software AG 的 Tamino。现在，XML 数据库可以用于多种不同的场景，比如内容管理和供应链管理系统、文档管理、出版以及 SOA 支持。

XRX 架构

除了刚才已经提到的用法，XML 数据库还是所谓“XRX 网络应用架构”的中心，这个概念也是近年来逐步走红的。这种架构是“对称的”有时也称为是“零转换”的，因为它在应用的每层都使用 XML。XRX 是 XForms、REST 和 XQuery 三者的首字母缩写。

在客户端，XRX 使用 XForms，这是 W3C 建议的 HTML 表单的 XML 格式替代品，XForms 可以描述表单所需的各种控件，不过它不强制要求最终的表达形式。也就是说，XForms 可以在网页中，也能在其他应用里展现，比如 Open

Office 或是 Lotus 文档。它是通过“模型-视图-控制器”（MVC）设计来达到这种灵活性的，还允许使用 XML schema 验证、页面内数据刷新等高级特性。目前，还没有浏览器原生支持 XForms，不过 Firefox 已经有插件支持 XForms，也有很多程序库可以在服务端将 XForms 转换成 XHTML。

在中间层，XRX 使用 RESTful 服务，甚至有时直接暴露数据库的 RESTful 接口；在后端，XRX 使用原生 XML 数据库存储，并用 XQuery 查询文档，这些数据库通常会提供一个 RESTful 接口。

XML 数据库已经被部分证明非常有用，因为它们允许开发者对于某些的 XML 文档直接使用 XML 工作。之前我们提到开发者使用面向对象语言与关系型数据库时面临了“阻抗不匹配”情况，在数据层直接使用 XML 的需求也与此类似。

XML 数据库有一类核心功能：允许你存储和查询 XML 文档。虽然它们通常都不直接使用“原生”格式进行存储，但开发者们可以通过 XML 的 API 来访问，这就好像后端使用了 XML 格式存储一般。它们的功能特性包括如下几项。

- 使用 XML 友好的查询机制，如 XPath 和 XQuery。XPath 是在文档中对不同数据项寻址的机制，如元素和属性。XQuery 提供了一个稳固的 FLOWR 形式查询机制（FLOWR 是 For、Let、Where、Order、Return 五种语句的首字母缩写）。
- 直接在应用中使用 XML 的时候可以获得性能提升。某些使用 XML 的应用不得不将文档映射到关系型数据库当中，但是如果省略映射的过程直接使用 XML 数据库将有很多好处。比如，XML 文档一般代表层次化数据结构，而此结构可能难以映射到关系模型之中。
- 可以灵活地访问数据。XML 数据库让你常常可以用 DOM、JDOM、SAX API 和 SOAP 协议来访问数据。这些协议都有自己的专长，你不必受限于 SQL 的单一的查询机制。
- 快速的全文搜索。
- 关系型数据库里所熟悉的功能，如跨文档集合的 join、用户定义函数、元数据和数据的搜索等。
- 灵活使用 XML 协议栈，如在展现层使用 XForms。
- 其他特性，比如存储非 XML 文档 [如纯文本（非结构化）文档]。

如果你还不熟悉 XPath 这种在 XML 文档里查找数据的方法，看看下面这个 XML 示例文档：

```
<catalog>
  <plays>
    <play name='Hamlet'><price>5.95</price></play>
```

```
    <play name='King Lear'><price>6.95</price></play>
  </plays>
</catalog>
```

对于上面这个 XML 文档，下面的 XPath 表达式的返回值应该是 6.95，这个结果是通过查找名称为“King Lear”的 `play` 元素里的 `price` 元素得到的：

```
//catalog/plays/play/[@name='King Lear']/price
```

有多个 XML 数据库的开源项目和商业产品可供选择。通常它们会使用两种存储机制之一：基于文本或是基于模型。基于文本的 XML 数据库通常将数据存储为一个大文本文件，或是字符型大对象（CLOB），甚至是二进制大对象（BLOB），它们存储在底层关系型数据库中，并提供转换机制。基于模型的 XML 数据库并不直接存储 XML 文档的文字内容，相反，它们把文档转化成内部的面向开发者呈现为 XML 文档的专有对象模型。这常常会把 XML 文档拆分成很多不同部分（各个元素、属性等），将它们分段存放到一个关系数据库中。

下面几节会简单浏览一些流行的 XML 数据库。

A.3.1 SoftwareAG Tamino

Tamino 是最早的 XML 原生数据库之一。它是一个成熟的商业产品，广泛支持你所期望的“企业级”数据库所需的各种功能，比如高可用。

A.3.2 eXist

eXist XML 数据库最初是 Wolfgang Meier 于 2000 年创建的个人项目，并一直持续积极发展至今。这是一个使用 Java 编写的开源项目。它提供了对很多机制的支持，包括 XPath、XQuery，以及 XInclude、WebDAV（分布式授权与版本）、用于安全的 XML 访问控制标记语言（XACML）、SOAP、REST 和 XML-RPC。eXist 还提供了一个易用且基于 Web 的用于查询的控制台。

A.3.3 Oracle Berkeley XML DB

Berkeley XML DB 是一个用 Java 编写的开源数据库，最初是一个哈佛的研究项目，目前由 Oracle 支持。Berkeley XML DB 可以嵌入到应用之中，可以作为一个 JAR 包放在应用里。它支持 C++、Java、XQuery、高可用和事务。Berkeley 数据库更加倾向于为开发者而非 DBA 设计，唯一与数据库交互的手段就是写代码，没有独立的服务器，也没有 SQL Server Management Studio 之类的图形化工具。可以用 Berkeley XML DB 混合存储 XML 文档和非结构化文档。

A.3.4　MarkLogic Server

MarkLogic 是由支持 XQuery 创建、读取、更新、删除（CRUD）操作，全文搜索，XML 检索和事务的 XML 数据库所支援的服务器。它支持使用 XML 文档或 JSON 的 REST 接口。MarkLogic 是一个商业产品，但对于小项目和非盈利组织，可以使用一个免费的社区授权。

A.3.5　Apache Xindice

Apache Xindice 项目也是早期 XML 数据库之一，开始于 2001 年。按照设计，它仅用于小型或中等尺寸的文档。自从 2007 年，它的上一个版本——1.1 版本发布之后，就没有积极的维护了，版本 1.2 的工作已经进行了很多年了。

A.3.6　小结

实际上还有很多其他的 XML 数据库，包括 TigerLogic、MonetDB、Sedna 等。不过，在这里介绍 XML 数据库的重要目的是，为随后介绍更新的面向文档的数据库做一些铺垫。更确切地说，这些数据库的最大亮点就在于，从这里可以看到通过将数据库映射到应用特定架构的需求而实现的优势，而非草率地认为关系型数据库可以包治百病。

当然，如果你只是存储小型 XML 文档，而且应用不需要文档集合，那么可能使用 XML 数据库也不会获得什么性能上的改观。

A.4　面向文档的数据库

在关系型数据库中，数据存放在表中，也有可能，数据会被反复“拆散”，以便使用关系键值。然后，写出复杂的查询语句，以便通过查询在行列二维的表格中重组得到所需要的关系型数据。

而面向文档的数据库一般会有这样几个优点。

- 文档数据库的基本存储单位就是一整篇文档。一篇文档可以存储任意数量的不限长度的域，每个域可以存储多个值。在这方面，文档数据库与关系型数据库的不同就在于，后者每个记录的所有域都必须有值。
- 在一个面向文档的数据库中，不需要像 RDBMS 中那样，在没有值可存的时候存储“空”域。这就节省了数据库的空间。
- 它们都不需要 schema，形式上非常随意。
- 可以对每个文档单独设定安全级别。

- 它们通常都支持全文检索。少数 RDBMS 也能提供这个功能，但在面向文档的数据库中就非常普遍了。

那么，到底什么是“文档”呢？可以是文本，存储为 JSON 格式（参考“何谓 JSON”）；也可以是 XML，前面已经单独讨论了使用 XML 的文档数据库；还可能是 YAML 文档（大部分 JSON 文档都可以被 YANL 解析器解析）；还可能是其他格式，因为有太多的其他选择。任意两个文档数据库的实现技术都不相同。举例来说，CouchDB 把数据存储为 JSON，而历史悠久的 Lotus Notes 则使用它自己的内部格式。

何谓 JSON

JSON 是 JavaScript Object Notation（JavaScript 对象标记）的缩写，是一种可以替代 XML 的数据交换格式。一个简单的用于存储地址簿里的联系人信息的 JSON 文档大致会是这样的：

```
{
   "contact": {
     "fname": "Alison",
     "lname": "Brown",
     "address": {"street": "301 Park Ave", "city": "New York", "state":
"NY", "zip": "10022"},
     "phone": [
         { "type": "mobile", "number": "480-555-5555" },
         { "type": "home", "number": "212-444-4444" }
     ]
   }
}
```

同样的信息，用 XML 表述应该是这个样子的：

```
<contact>
    <fname>Alison</fname><lname>Brown</lname>
    <address>
      <street>301 Park Ave</street><city>New York</city>
      <state>NY</state><zip>10022</zip>
    </address>
    <phone type="mobile">480-555-5555</phone>
    <phone type="home">212-444-4444</phone>
</contact>
```

JSON 只支持少量数据类型，没有 XML 提供的那么丰富。JSON 支持的数据类型包括数字、Unicode 字符串、布尔、数组、对象和空。

在上面这个 JSON 文档中，你可以看到一个带有几个属性的联系人对象。名和姓以字符串形式存储。地址属性是一个对象，因为它也定义了自己的属性。最

后，电话属性有一个数组值，使用方括号表示，里面的类型属性是重复出现的。JSON 的互联网媒体类型是 application/json。

JSON 大约在 1999 年就出现了，但直到近几年，当开发者已经厌倦了臃肿的 XML 以及它难用的工具和 API 时，JSON 才迅速流行起来。一个简单的事实就是，JSON 文档通常比 XML 文档短 30%，Google 在 2006 年开始使用 JSON 作为其 GData 协议的格式，这在很大程度上提高了 JSON 的知名度。

相对于 XML 文档，JSON 文档更加轻量级，虽然 XML 在很多平台上都有广泛的工具支持，但 JSON 文档非常简单，交互时也需要更少的工具支持。对于某些应用来说，JSON 的一个潜在的缺点是，没有 XML schema 那样的方式可以直接验证文档结构。

在直接表示数据的时候，或是在文档数据库和分布式哈希表的例子中，都经常使用 JSON，所以它很值得单独介绍，在本书中，也经常见到 JSON 的身影。

你可以把面向文档的数据库当做是一些键–值集合，把它们看做是稍后在 A.6 节讨论的“键–值存储和分布式哈希表”的前身。这里有一个简单的 JSON 文档：

```
{
 "title": "I Heart LolCatz",
 "author: "Inigo Montoya",
 "ts": Date("31-Dec-99 11:59"),
 "comments": [{
  "author": "Robert Zimmerman",
  "comment: "I'm just a song and dance man"}, {
  "author": "Rogers Nelson",
  "comment: "I'm just a song and dance man"}
 ]
}
```

看起来似乎很简单，而考虑一下如何用关系数据库里的表来表达一个如此简单的数据结构，以及如何查询，可能就不是那么容易了。但在面向文档数据库里，存储的就是这个文档，查询也会非常简单。

A.4.1 IBM Lotus

Lotus 最早的版本发布于 1989 年，可能是所有面向文档的数据库的灵感来源，包括 CouchDB 和 MongoDB。Lotus 是一组用于协作的软件产品套件，包括用于 email、讨论和日历的 Lotus Notes 和 Domino，以及用于即时消息的 Lotus Sametime 及其他组件。

- **网站**：http://www.ibm.com/software/lotus
- **面向**：文档
- **创建**：Lotus 的首个版本发布于 1989 年。截止到本书写作时的最新版本是 2010 年 3 月发布的 8.5 版本。
- **schema**：不需要 schema。文档（“notes”）存储在称为 Notes 存储文件（NSF）的自有格式内，但可以看做是一种具有与 JSON 类似建模目的的文档。
- **客户端**：最新版本的最终用户客户端是基于 Eclipse 的。要与 Domino 数据库交互，可以使用 C、C++ 或是 Java 的 API。Notes 的数据库是非关系型的，但可以使用一个 SQL 驱动来访问它，而且 Domino XML 语言还提供了全部数据的 XML 视图，你可以使用 XML 工具处理数据。
- **CAP**：Lotus 可以集群化并进行副本复制。
- **产品应用**：Lotus 在很多企业中都作为员工协作工具使用。

A.4.2 Apache CouchDB

作为一种数据库，CouchDB 可能是和 Lotus Notes 最为相似的了。这并不奇怪，它的创始人 Damien Katz 在决定开始创建这个项目之前就是在 IBM 从事 Lotus Notes 相关工作的，后来开始感到它可以作为一种“为 Web 而生的”数据库，才开始了 CouchDB 项目。CouchDB 中的文档不需要使用相同的 schema，可以通过视图进行查询，视图是通过 JavaScript 函数构成的。

CouchDB 最有趣的特性之一是多版本并发控制（MVCC）。MVCC 意味着读操作将不会阻塞写操作，写操作也不会阻塞读操作。为了支持这个特性，所有的写操作都作为追加写加入到文档之中，这样就很难破坏数据文件了。这个实现和 Cassandra 有些相似，但使用一个纯追加写模型意味着文件可能会很快就增长到很大，需要一个后台进程来进行压紧操作。



如果你希望更多地了解 CouchDB，可以参阅 O’Reilly 的《CouchDB 权威指南》（*CouchDB: The Definitive Guide*），由 J. Chris Anderson、Jan Lehnardt 和 Noah Slater 合著。

- **网站**：http://couchdb.apache.org
- **面向**：文档
- **创建**：开始于 2005 年，2008 年成为 Apache 孵化器项目。
- **实现语言**：Erlang
- **分布式**：是，离线时数据也可以被用户和服务器读取或更新，在之后，所有的变更都可以进行双向同步。

- **schema**：不需要 schema。文档可以全部以 JSON 格式存储。每个文档有唯一的 ID。
- **客户端**：RESTful 的 JSON API 允许任何可以发起 HTTP 请求的语言访问。
- **CAP**：最终一致性。具有副本复制机制，用于同步不同节点上的多份数据副本。与很多关系数据库类似，Couch 提供了 ACID 语义。
- **产品应用**：CouchDB 在本书写作时仍然没有完成 1.0 发布，但在多个社交网站和软件应用中已经有了对应产品。参考 http://bit.ly/dn73DY 中的产品应用列表。
- **附加特性**：支持 MapReduce、增量副本复制和容错性。带有 Web 控制台。

A.4.3 MongoDB

MongoDB 可能和 CouchDB 最为相似。它旨在融合键值存储、文档数据库、对象数据库和 RDBMS 的精华之处。具体地说，它和键值存储一样自动分片，允许使用基于 JSON 的动态 schema 文档，并提供关系数据库形式的丰富的查询语言。



如果你希望更多地了解 MongoDB，可以参阅 O'Reilly 的《MongoDB 权威指南》（*MongoDB: The Definitive Guide*）[1]，由 Kristina Chodorow 和 Michael Dirolf 合著。

- **网站**：http://www.mongodb.org
- **面向**：文档
- **创建**：由 Geir Magnusson 和 Dwight Merriman 在 10gen 开发。
- **实现语言**：C++
- **分布式**：是
- **schema**：存储 JSON 风格的文档，可以使用动态 schema。
- **CAP**：MongoDB 对所有分片都使用单主节点的方式，可以达到完全一致性。
- **产品应用**：MongoDB 的用户包括 SourceForge、Bit.ly、Foursquare、GitHub、Shutterfly、Evite、纽约时报、Etsy 和很多其他公司。
- **附加特性**：支持 MapReduce。有一个很简洁的 Web 界面，可以让你在浏览器里使用 JavaScript shell 来尝试 MongoDB。可以在这里访问：http://try.mongodb.org。

A.4.4 Riak

Riak 是一个基于 Amazon Dynamo 的混合型数据库，既是面向文档的，也是分布式键–值存储的。Riak 具有容错性，可以线性扩展，是为互联网应用而设计的。Riak 和 Cassandra 有些相似，也没有中心控制器，自然也就没有单点失效。

译注 1：本书已由人民邮电出版社出版。

Riak 的设计中有三个基本元素：桶（bucket）、键、值。数据组织在桶中，桶大致相当于一个平坦的命名空间，用于在逻辑上组织键值对。这和 Google 的存储系统在设计和命名上都很相似。

Riak 的实现者 Basho Technologies，同时提供了商业版本和开源版本。

Riak 可以在大部分 Unix 类操作系统上运行，但不支持 Windows。

- **网站**：http://wiki.basho.com
- **面向**：文档与键值存储
- **创建者**：麻省剑桥的 Basho Technologies，这个公司由 Akamai 的架构师于 2008 年创建。
- **实现语言**：主要是 Erlang，部分使用 C 和 JavaScript
- **分布式**：是
- **副本复制**：可以在桶级设置副本复制机制。
- **schema**：Riak 是无 schema 的，不使用特定的数据类型。和键对应的值是对象。所有的数据以不透明的 BLOB 形式存储，所以可以在 Riak 中存储任何类型的数据。
- **客户端**：Riak 提供了三种主要的交互方式：JSON/HTTP 的 API；Erlang、Python、Java、PHP、JavaScript 和 Ruby 语言的驱动；还有一个 Protocol Buffers 客户端接口。Protocol Buffers 是一个 Google 的高速 RPC 项目，之前是 Google 内部使用的，现在可以在 http://code.google.com/p/protobuf/ 访问。
- **CAP**：Riak 和 Cassandra 非常类似，这个数据库也允许用户来调整一致性、可用性和分区奈受性的级别。
- **产品应用**：Riak 的客户包括 Comcast 和 Mochi Media。
- **附加特性**：Riak 可以和 MapReduce/Hadoop 集成。Riak 的商业版本称为 Enterprise DS，还支持跨数据中心的同步（开源版本只支持单数据中心内同步），拥有一个 Web 控制台，并支持简单网络管理协议（SNMP）。

A.5 图数据库

图数据库是关系数据模型的又一种替代模型。你可以把图想成一个网络。图数据库的数据不是存储在表或者列里，而是使用三个基本元素来代表数据：节点、边和属性（property）。在图数据库中，*节点*是一个独立的对象，不依赖于其他任何东西；*边*依赖于两个节点；*属性*就是节点或边的一些属性内容。比如，个人节点（personal）可能会有一个姓名属性和一个 email 属性。节点和边都可以拥有与之关联的属性。

图 A-1 示意了一个有五个节点和五条边的图数据库。边可以描述多种关系，但在这

个例子中，边只有一个名称属性。可以看到，有一条边是双向的。例子里的每个节点都有一个属性（name），但它们都可以有多种不同附加属性。图数据库的一个很好的特点是它们是“白板友好的”，也就是说，图数据模型看起来和我们把事物搬到白板上的样子很相像，不需要任何多余的转换，就可以让数据符合数据库的约束条件。

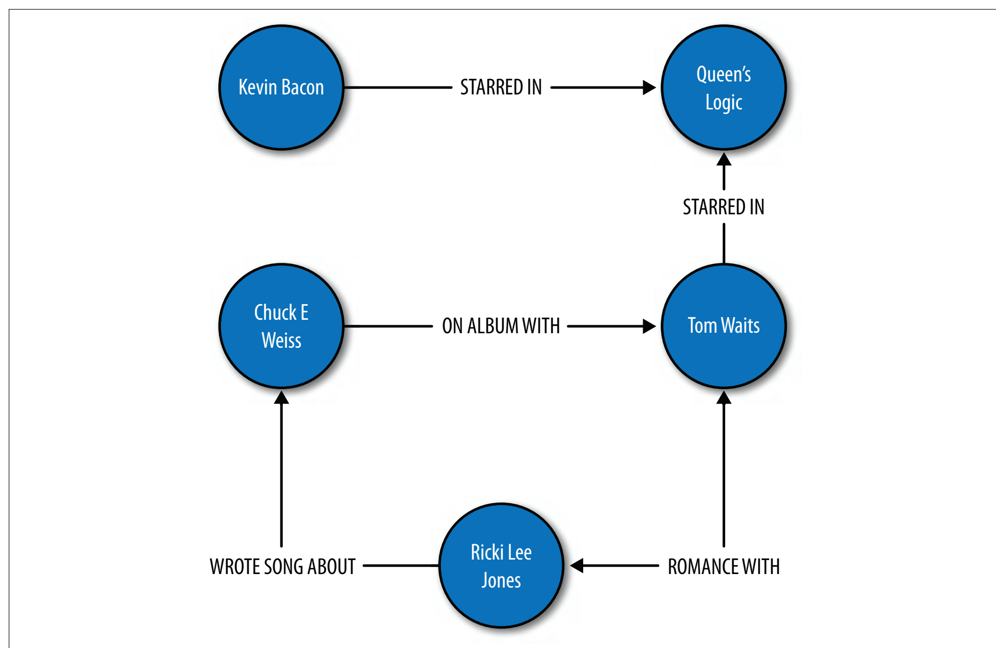


图 A-1：图建模关系

图数据库和其他的非关系型数据库的不同在于，比如键–值存储，在图数据库中，不仅仅是节点，边也是一等公民。这就是说，尽管很多我们使用过的编程语言和数据库允许表达一种关系，但这些关系本身都是间接的（比如通过外部键值或是指针）。但在图数据库中，关系和节点是平等的，因为节点间的关系在某些应用场景中可以占据中心地位。例如，图数据库在近年来变得很流行，因为它们刚好适用于社交网络领域，并可以很好地适用于 Web 2.0 社交网络中的各种查询，因为关系刚好就是社交网络的精髓所在。图数据库的第二个重要的而且处于增长之中的适用场合是语义网络，在语义网络中，谓词和主语、宾语在三者具有同样的地位，构成三位一体。

在过去 15 年中，有两个关键趋势决定了图数据库作为一种重要的数据存储形式的崛起。其一是数据在数量上的增长，其二是数据之间的关联的增长，这两个趋势就使得图数据库成为一个有吸引力的选择。

以图 A-2 所示的简单的时间线为例。从 20 世纪 90 年代初开始，文本文档和相互链接的简单网页展示了互联网的大量内容。这些文档是直接存储的，而且易于使用关系型数据库来生成。之后在本世纪早期，RSS 订阅、博客和维基开始崛起，自动化和引用链接的新方式威胁到了关系模型。到了 2005 年，Web 2.0 开始崛起（或者按照 Sun 的 Jonathan Schwartz 的定义，称为用户参与的时代），分众、标签云和分类开始走入我们的视野，所有这些既为机器的资源消耗和接口而优化，也为人们使用而优化。我们开始不再把网络看做是像一页页的杂志那样，通过动态地从一个关系数据库取出相关数据组织而成，现在，我们觉得网络不仅是一种数据的表现方式，更是一种概念的表达。高速互联网基础设施的逐渐完善，使得社交网络站点茁壮成长。很多人突然被联系到了一起，互联网为他们提供了新的交互方式，而不仅仅是阅读。

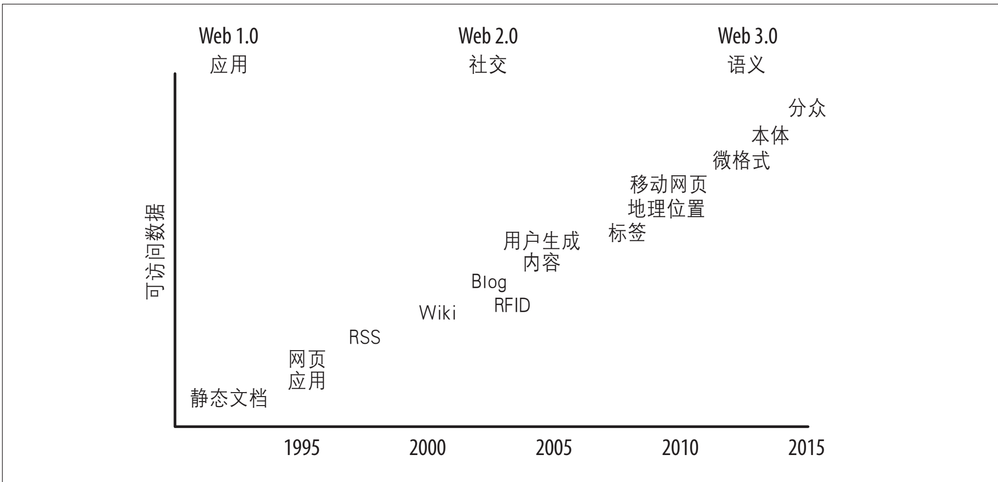


图 A-2：管理大规模数据的需求正在增加，而且还将继续

当互联网开始构建重要的语义层时，伴随而来的就是数据量的爆炸，这时，最大的互联网巨头开始被迫寻找关系型数据库的替代产品。一些语义网络的研究者和爱好者最近发表看法认为，我们正在进入一个新的 Web 3.0 时代，这里，RDF（资源描述框架）、微格式和本体（ontology）将支持起超连结（superconnected）的“数据网络”高潮，这将是世界上的所有知识库构成的一个巨大的图。这里，图数据库意义更加显而易见，因为它们直接支持节点间的语义关系的概念。

与许多面向文档的数据库类似，图数据库一般不要求 schema 的形式，它允许应用随着数据集的增长和变化来调整数据结构。因为它不是关系模型，join 在这里没有必要，图数据库随着数据集的增长，可以提供更高效的查询。

相对于 RDBMS，图数据库最本质的优点是没有所谓阻抗不匹配的问题，你可以把对象按照应用使用它们的方式存储到数据库中，就像白板上画出它们一样，更为直接、易于理解、轻松建模。像键-值存储与面向文档的数据库一样，图数据库允许存放半结构化数据，并自然地根据新发现的关系和属性来发展 schema。

如果你对社会化网络和语义网络应用中的图数据库感兴趣，还可以研究更多在此未一一讨论的图数据库，包括 Dex、HypergraphDB、Unfogrid 以及 VertexDB。我还建议你看看 http://wiki.github.com/tinkerpop/gremlin 中的 Gremlin 项目。Gremlin 是一个为进行查询、图分析和处理图数据库设计的开源程序语言。Gremlin 可以从 Java 虚拟机中运行，它的实现遵从 JSR223。

这里只介绍两种图数据库，但是，如果你是一个 Hadoop 使用者，还可以看看 Hama 项目，本书写作时它还是孵化器项目。Hama 是一个在 Hadoop 上构建的包，支持大量矩阵和图数据。参考这里 http://incubator.apache.org/hama。Google 还有一个称为 Pregel 的项目，他们内部使用这个项目很多年了，并且可能会开放源代码。关于 Google 的 Pregel 相关的信息可以看这里：http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html。

A.5.1 FlockDB

2010 年 4 月，Twitter 宣布他们在 GitHub 上开源了新的图数据库，称为 FlockDB。他们创建了 FlockDB 来存储 Twitter 中的跟随者（follower）的邻接列表，这样就可以了解到谁跟随了谁，以及谁封锁了谁。FlockDB 可以水平扩展，其设计应用场景的特征是：在线的、低时延和高吞吐量。Twitter FlockDB 集群存储了超过 130 亿条边，服务峰值流量达到每秒钟 2 万次写操作和 10 万次读操作。

- **网站**：http://github.com/twitter/flockdb
- **面向**：图
- **创建**：2010 年由 Twitter 创建
- **实现语言**：Scala
- **授权方式**：Apache 许可证（第 2 版）
- **分布化**：是
- **schema**：FlockDB 的 schema 非常简单直接，因为 FlockDB 并不想解决所有的数据库问题，仅仅希望解决那些和 Twitter 的关系图与数据集尺寸有关的问题。FlockDB 的图里包含的条目有四个属性：一个源 ID、一个目的 ID、一个位置和一个状态。
- **客户端**：FlockDB 使用 Thrift 0.2 客户端，Twitter 还写了一个 Ruby 的前端，提供了更丰富的接口。

- **副本复制机制**：有
- **底层存储**：MySQL
- **产品应用**：Twitter
- **附加特性**：FlockDB 允许快速对包含上百万个条目的查找结果集进行分页，支持对边进行归档和恢复操作。它松耦合地使用了 Kestrel，作为可靠的消息队列服务，然后随机选择服务器来写入数据，所以没有服务器间通信（没有集群操作、没有组播等）。

A.5.2 Neo4J

Neo4J 是一个兼容 ACID 属性的图数据库，特别为快速的图遍历而优化。它支持事务，支持 JTA/JTS、两阶段提交、死锁检测以及 MVCC。Neo4J 早在 2003 年就有产品应用了，这让它成为了这里比较老的一个存储系统。在一个 JVM 上，Neo4J 可以扩展到数十亿个条目（节点、边、属性）。

Neo4J 可以作为 JAR 包嵌入到应用之中，很容易就可以设置并运行 Neo4J，并且可以灵活地在应用之中使用。

- **网站**：http://neo4j.org
- **面向**：图
- **创建**：由 Neo Technologies 于 2003 年创建，并开始产品应用。版本 1.0 发布于 2010 年 2 月。
- **实现语言**：Java
- **许可证**：开源许可证为 AGPLv3，使用商业许可证可以获得更多的特性。
- **分布化**：Neo4J 是半分布化的，可以使用 RMI 进行远程通信。注意：免费版本的 Neo4J 是不能分布到多台机器上的。
- **schema**：Neo4J 是一个由无 schema 的节点、边和可选属性构成的图。
- **客户端**：有多种选择：Neo4J 可以作为一个 REST 服务器来启动，这样就可以使用简单的 HTTP 操作、用 JSON 进行操作；Neo4J 还有一个 shell 客户端界面。Neo4J 有各种语言的支持，包括 Java、Python、Ruby、Clojure、Scala 以及 PHP。可以通过 Neo4j.py 来查看 Jython 和 CPython 接口，通过 Neo4j.rb 查看 JRuby 支持。Neoclipse 是一个 Eclipse IDE 的插件，为图提供了图形化展现界面，并有一个 Grails 的接口。
- **副本复制机制**：截止到本书写作的时候，Neo4J 的副本复制功能仍然在开发中。它的主从副本复制的设计基于 MySQL 中使用的副本复制机制，可以对任意从实例写数据，锁协调和改变分布机制等则由主实例控制。你可以调节合适的一致性级别，通过要求 Neo4J 同步写到主从节点上来达到强一致性，也可以在写操作时使用异步

传播到从节点的方式，通过最终一致性来换取更好的性能。

- **存储**：客户磁盘存储
- **产品应用**：box.net、ThoughtWorks
- **附加特性**：因为 Neo4J 是图数据库，所以可以很好地应用在语义网络应用之中。允许执行 SPARQL 协议和 RDF 查询语言（SPARQL）与资源描述框架进行交互，并作为部分的网络本体语言（OWL）存储。

Neo4J 可以和 Apache Lucene/Solr 集成，以存储外部索引来进行更快的全局检索。索引在分布式数据库中可以看做是一个字典——从一个键指向一个值的直接指针。

Neo4J 的版本 1.1 将会带有一个事件框架。

A.6 键-值存储与分布式哈希表

在关系模型中，我们首先考虑的是，域要求使用什么样的表，然后考虑如何将表范式化，来避免数据重复。表、表中定义的列以及表之间的关系就是我们的 schema。

但是，在键-值存储中，通常不需要预先定义这样的 schema。域在这里成为了桶，你可以向桶中放入数据项，数据项是包含一组属性的键值。所有数据都与键值相关联，按照键值存储在一起，这和关系型数据库引以为傲的范式化模型形成了鲜明的对比：数据常常是有重复的。虽然键-值存储有一些变种，而且，有一些概念和列数据库有些重叠。

另一个对比关于建模。使用关系型数据库工作时，我们倾向于仔细考虑 schema，确信我们要问数据库的所有问题都可以解答。因为这些问题——查询——在模型中是次要问题，它们可能会非常复杂。你肯定看到过那些使用了多个 join、子查询、聚集函数、临时表等的复杂 SQL 语句。但在列状模型中，我们倾向于首先考虑查询，这些查询帮助我们决定了如何设计桶。列数据库中的假设是，为了保证数据库可用，我们需要数据副本，因为磁盘空间很廉价，所以数据重复完全可以接受。

数据完整性是另一个差异点。数据完整性是指应用中数据的完全和一致性的程度。关系型数据库具有保证数据完整性的能力，比如主键（保证数据项的完整性）以及外部键约束（保证引用完整性）。但在键-值存储之中，保证数据完整性的责任完全交给应用完成。

考虑一个例子：你在数据库中存储了客户和订单信息。在关系型数据库中，必须在数据库中定义引用键值来允许 join 表，以及进行诸如查看一个特定用户所有订单这样的操作。虽然在键-值存储中也能这么做，但通常不需要在数据模型中定义任何关

系信息，应用会在用户进行操作的时候维护数据完整性，比如在决定删除某个用户的记录的时候。

对键–值存储的一个苛责在于，当需要扩展到数十亿条记录的时候就会非常困难，但这种需求仅仅是在非常大的社交网站才会有。对这种情况的建议是，键–值存储从本质上意味着应用可以把数据库看成一个单一的、庞大的、可以全局访问的哈希表，这本身就难于维护，编程效率也很难保证。

如今有很多种可供选择的键值存储，包括 Tokyo Cabinet、亚马逊的 SimpleDB 以及微软的 Dynomite。

A.6.1 亚马逊的Dynamo

Dynamo 是亚马逊内部使用的键值存储系统。虽然开发者们无法使用这个系统，但它仍然非常值得讨论，因为 Dynamo 和 Google 的 Bigtable 一样，都是 Apache Cassandra 的很多设计决策的来源。

2007 年 10 月，亚马逊的 CTO Werner Vogels 在 ACM 发表了一篇名为“Dynamo：Amazon’s Highly Available Key-value Store”（亚马逊的高可用键值存储）的白皮书。这篇论文现在还可以在他的博客“All Things Distributed”（一切皆分布）上找到，链接为 http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf。这篇论文技术性很强，但非常清晰、简洁，并且写得很精彩。我将在这里简单总结它的主要观点。

Dynamo 和很多本附录中描述的系统一样，诞生于现实生产环境中的需求，包括持续增涨的性能压力、服务水平协议（SLA）要求、高负载和节点故障下的持续可用要求、各种故障的平滑处理，以及水平扩展能力。这样，遵照 CAP 理论，Dynamo 和 Cassandra 一样，是一个高可用和最终一致性系统。这些系统的故障处理被认为是一种“不应该影响可用性和性能的常态”。在 Dynamo 里，这是通过牺牲部分的一致性获得的。

Dynamo 用于亚马逊的购物车，当然，一致性对于亚马逊是非常重要的。但对于基于 Web 的购物车，实际并不存在竞争的读者，所以这个折中是比较值得的。虽然一致性不是这个系统的主要考虑焦点，但它的一致性还是可调的，所谓“最终”是有点用词不当的。

和 Cassandra 一样，Dynamo 的一致性是一个可配置的属性，它允许用户来确定一个副本数，响应数必须要达到这个副本数才可以认为操作是成功的。要做到这点，所有副本之间的通信是通过称为“gossip”的对等通信协议（P2P）来实现的，本书中

已经把这个概念作为 Cassandra 的本质特征做了深入的讨论。

Dynamo 架构的需求非常清晰。为了支持高可用的模型，团队决定调低一致性“旋扭”。重复一次，这在他们的应用场景中是完全可以接受的。他们还需要一个非常易于使用的查询模型，所以数据使用唯一的键值来引用，存储在一个简单的字节数组里。这就无须使用复杂的 schema 设计了，也允许亚马逊将更多精力放到他们的主要目标——低时延和高吞吐性能的优化上面了。

要达到一个可以接受的一致性级别，Dynamo 必须实现某种版本机制，以让副本知道，哪个节点有最新（有效）的数据副本。它使用了一种称为向量时钟的机制，每个进程维护一个数值引用，指向它所知道的最新事件。Cassandra 和 Dynamo 在架构上的另一个共同之处是有提示切换。

本节总结了 Amazon Dynamo 论文的主要内容，帮你理解其架构目标和特性。虽然我非常推荐你阅读 Dynamo 的论文，但也要注意，Cassandra 是在走自己的路，所以不要理所当然地认为 Dynamo 的一切描述都适用于 Cassandra。简单地说，Cassandra 在一致性和分区耐受性方面继承了 Dynamo 的设计，而在数据模型方面则基于 Bigtable。

A.6.2　Voldemort项目

Voldemort 开始是一个 LinkedIn 的内部项目，用于解决他们所面对的简单数据在可扩展性需求下的分区问题，与 Cassandra 在 Facebook 启动的原因颇为类似。Voldemort 是一个分布式的、极简单的键−值存储系统，基于 Amazon 的 Dynamo 和 Memcached。

LinkedIn 的 Jay Kreps 提到的性能数据大约是一个客户端、一个服务器的环境下，每秒 2 万次读操作、1.7 万写操作。

- **网站**：http://project-voldemort.com
- **面向**：键−值存储
- **创建**：2008 年，由 LinkedIn 的数据与分析团队为解决应用的实时性问题而创建。
- **实现语言**：Java
- **分布式**：是
- **schema**：Voldemort 的主要设计目标是高性能与高可用性。所以，这个数据库仅支持最简单的 schema。下面这些是仅有的查询方式：`value = store.get(key)`，`store.put(key,value)` 和 `store.delete(key)`。因为 Voldemort 允许用 JSON 指定 schema，所以它支持 JSON 支持的数据类型。

- **客户端**：和 Cassandra 类似，Voldemort 允许可插拔的接口方式。按照 Voldemort 的网站上的说法，它支持可插拔的序列化机制，并集成了 Thrift、Avro 和 Google Protocol Buffers 的支持。
- **副本复制机制**：数据自动地在多台服务器间进行副本复制，这是可配置的。
- **底层存储**：Voldemort 允许使用可插拔的磁盘存储机制，可以使用 BerkeleyDB 或是 MySQL。
- **产品应用**：LinkedIn
- **附加特性**：可以与 Hadoop 一起使用。

A.6.3 Redis

Redis 不是一个“普通的”键–值存储，因为它还支持多种不同数据结构的值，比如二进制安全字符串（不包含空格或换行的字符串）、列表，以及二进制安全字符串集合、有序集合，其中有序集合包含一个用于排序的浮点数分数。

2010 年 3 月，VMWare 接手，成为 Redis 项目的赞助者。

- **网站**：http://code.google.com/p/redis
- **面向**：键–值存储
- **创建**：创建于 2009 年
- **实现语言**：ANSI C
- **分布化**：没有分布化和容错性。
- **schema**：键–值存储，使用 `server:key-name` 来存储与取出数据。
- **客户端**：Redis 支持多种客户端，通常通过外部的库，支持 Ruby、Python、Twisted Python、Erlang、Tcl、Perl、Lua、Java、Scala、Cloure、C#、C、Haskell，以及 Google 新的 Go 语言。
- **CAP**：最终一致性
- **是否开源**：是。托管在 Google Code 项目中。这里有一个很整洁的页面（基于 MongoDB 的教程），提供了一个 Redis 教程，并允许你直接在浏览器里使用 JavaScript 来尝试 Redis。来 http://try.redis-db.com 尝试一下吧。

A.7 列数据库

在列数据库中，数据围绕列而非行来组织。这样，可以对某些应用的负载更加优化，特别是数据仓库和分析类应用，因为它们在计算时需要聚集大量的相似数据。列数据库（或说面向列的数据库）特别适合那些对大数据集进行查询的在线分析处理（OLAP）应用。

为了优化磁盘空间和IO消耗的时间，在列数据库里，数据存储的工作方式有些许不同。例如，列数据库允许写一条记录时，只写很多列中的一列，并且只有这列会占用实际空间。这和RDBMS非常不同，在RDBMS里，空值也不是零开销的。你可以把RDBMS想象成一个工作表，每行的同一列都占用同样的空间，为了维持表格数据结构的形状，即使没有东西也要维护一个空值在那。这个模型对列数据库根本就不适用，因为没有“空”值存在。列数据可以想象成是标签：值可以是任意长的，并且列的名字和宽度都不做预设。

列数据库经常要求数据的类型是一致的，这样可以更好地进行数据压缩。

列数据库最早大约出现在20世纪70年代早期。Sybase IQ就是最早的列数据库之一，很多年来，都只有商业的列数据库。

不过，近几年的项目（主要是开源项目）才是我们讨论的NoSQL的一部分，这些数据库演进自基本的键-值存储，但拥有更为丰富的数据模型。你可以把这些列数据库看做是多维的键值存储或哈希表，它们不仅支持简单直接的键值对，还允许使用“列族”这样的数据模型来帮助组织各列，提供更丰富的模型。这些列数据库包括Google的BigTable、HBase、Hypertable和Cassandra。

Google的Bigtable是现代列数据库的鼻祖。它是一个内部自用的系统，但是Google发表了几篇论文来介绍它的设计，后面所有讨论的列数据库的实现都紧随Bigtable的设计。就Cassandra而言，它也从Bigtable继承了一些关键理念。

A.7.1 Google Bigtable

在Google内部，Bigtable用于用户数据库，它的设计可以扩展到PB级别。Google在2006年发表的论文《Bigtable：结构化数据的分步式存储系统》（Bigtable: A Distributed Storage System for Structured Data）中介绍了Bigtable。在论文中说明了这个项目的目标为：“广泛适用、可扩展、高性能与高可用。”Bigtable在Google内部作为底层存储，使用非常广泛，支持了超过60个项目，包括Gmail、YouTube、Google Analytics、Google Finance、Orkut、个性化搜索和Google Earth。Bigtable运行在Google文件系统（GFS）之上。

理解Bigtable非常有意义，至少在某种程度上，因为它的很多特性和设计决策都是Cassandra的蓝本。虽然Cassandra在一致性和分区耐受性方面是基于Amazon Dynamo的设计的，但它的数据模型和Bigtable更为接近。例如，Cassandra从Bigtable中借鉴（也有一些修改）了SSTable、memtable、Bloom filter和压紧（compaction）的实现（参考词汇表里有这些名词的定义，它们在本书的其他章节里

有详尽的介绍）。从这个角度看，Cassandra 比 Dynamo 支持更丰富的数据模型，比简单的键值存储更为灵活和层次化，因为它支持稀疏、半结构化的数据。



我强烈建议你阅读 Google Bigtable 的论文，这是一篇非常优秀的文章。不过，需要注意的是，虽然 Cassandra 从 Bigtable 中拿来了很多关键理念，两者无论概念还是实现也不完全是一一对应的。比如，Bigtable 定义了主从节点，尽管 Cassandra 的数据模型和存储机制是基于 Bigtable 的，也有很多地方使用了相同的名词，但也并不总是这样。比如，Bigtable 的读写与 Cassandra 的实现接近但不一样，Bigtable 定义了 Tablet 结构，但在 Cassandra 里没有严格一致的体现，等等。你可以在 http://labs.google.com/papers/bigtable.html 找到 Bigtable 的论文。

Cassandra 在很多地方和 Bigtable 有差别，但是，两者重要的差别在于 Cassandra 使用了无中心的模型。在 Bigtable 里，主节点使用 Chubby 永久分布式锁机制来控制操作；而在 Cassandra 中，所有的节点都是平等的，没有中心控制，使用一个 gossip 模型来互相通信。

Bigtable 依赖于一个称为 Chubby 的分布式锁服务，它用 Chubby 做几件事情：保证任何时候至少有一个主节点的副本可用，管理服务器的系统引导、发现和死亡，存储 schema 信息。

- **网站**：无，但可以在 http://tables.googlelabs.com 查看一个相关的项目，称为 Google Fusion Tables。
- **面向**：列
- **创建**：Google 从 2004 年开始 Bigtable 的开发，论文发表于 2006 年。
- **实现语言**：C++
- **分布化**：是
- **底层存储**：Google 文件系统（GFS）。文件分割为 64 MB 的分块，通常只以追加的方式写文件，以提供最好的吞吐量。GFS 有一个根本原则，文件系统必须运行在大量廉价的普通服务器上，随时可能会出错，因此必须能够在这种场景实现可用性。GFS 有两类服务器：一个主节点和很多数据块服务器（chunkserver）。chunkserver 存储数据块文件，主节点则用来存储所有关于数据块的元数据，包括数据的存储位置等。显然，在存储方面，Cassaandra 与 Bigtable 有很多不同，因为 Cassandra 里，所有节点都是地位相同的，没有主节点对环做中心控制。
- **schema**：Bigtable 的数据模型是一个稀疏、分布化、多维的有序映射。允许用户存储比亚马逊 SimpleDB 更为丰富的数据，因为它支持列表类型。这个映射使用行键值、列键值和时间戳进行索引，值本身是无解释的字节数组类型。

- **客户端**：C++，查询有时也可以使用一个 Google 内部开发的脚本语言 Sawzall。最初 Sawzall API 不支持向数据库写值，但可以进行数据过滤、转换和汇总。MapReduce 既可以从 Bigtable 输出数据也可以输入到 Bigtable 中。
- **是否开源**：否
- **附加特性**：虽然你无法直接使用 Bigtable，但可以通过在 Google App Engine 创建应用来间接使用它。Bigtable 在设计时就考虑了用于 MapReduce 的算法。Bigtable 有好几个克隆产品，而 Hadoop 是一个 MapReduce 的开源实现。

A.7.2 HBase

HBase 是 Google Bigtable 的一个克隆，最早是和 Hadoop 一起使用的（实际上，起初是 Apache Hadoop 项目的一个子项目）。与 Google 的 Bigtable 底层使用 GFS 类似，HBase 为 Hadoop 提供数据库能力，允许使用它作为 MapReduce 任务的数据源和输出目的。与其他的提供最终一致性的列数据库不同，HBase 是强一致性的系统。

值得一提的是，微软也是 HBase 的一个贡献者，因为他们收购了 Powerset。

- **网站**：http://hbase.apache.org
- **面向**：列
- **创建者**：Powerset 在 2007 年创建了 HBase，其后捐给了 Apache。
- **实现语言**：Java
- **分布化**：是。可以单独运行 HBase，或伪分布化，或是使用完全分布化模式。伪分布化模式意味着运行多个运行在同一台服务器上的 HBase 实例。
- **底层存储**：HBase 构建在 Hadoop 分布式文件系统之上，类似于 Bigtable。
- **schema**：HBase 支持非结构化和部分结构化数据。要支持这些数据结构，HBase 用列族（这个概念同样出现在对 Apache Cassandra 的讨论中）来组织数据。可以使用行键值、列族、单元限定符和时间戳来寻址一个单独的记录，在 HBase 里称为"单元（cell）"。与 RDBMS 不同，在 RDBMS 之中，必须预先定义好表结构，而 HBase 允许只定义一个列族，然后单元限定符可以在运行时决定。这让应用可以非常灵活，并支持采用敏捷式开发方法。
- **客户端**：可以使用 Thrift、RESTful 网关、Protobuf（参见下面的附加特性）或一个可扩展的 JRuby shell 和 HBase 交互。
- **是否开源**：是（Apache 许可证）
- **产品应用**：Adobe 从 2008 年开始使用 HBase。Hbase 的用户还包括：Twitter、Maholo、StumbleUpon、Ning、Hulu、World Lingo、印尼的 Detikcom 以及 Yahoo。

- **附加特性**：因为 HBase 是 Hadoop 项目的一部分，它和 Hadoop 的结合非常密切。因为有一整套的便利类，所以如果你希望使用 HBase 作为存储系统，并在上面运行 MapReduce 任务，这非常方便。

HBase 的运行需要依赖于 Zookeeper。Zookeeper 也是 Hadoop 项目的一部分，是一个用于维护配置信息、在集群的节点间提供分布式同步机制的中心服务。虽然 HBase 使用 Zookeeper 会增加外部依赖，但这让集群的维护更加简单，而且简化了 HBase 的核心代码。

你还可以使用 Google 的 Protobuf（Protocol Buffer）API 代替 XML 来访问 HBase。Protobuf 是一种非常高效的数据序列化方法。相同的数据，它可以压缩到 XML 的一半到三分之一，比 XML 解析要快 20~100 倍，因为 protocol buffer 的在写入的时候进行编码。使用 Protobuf 可以让 HBase 工作非常快。Protobuf 在 Google 内部使用非常广泛，他们使用 Protobuf 在不同的系统之间传输接近五万种不同消息类型。可以从 Google Code 中访问这个项目 http://code.google.com/p/protobuf。

HBase 提供了一个基于 Web 的控制台用户界面，可以用于监控管理区域服务器（region server）和主服务器。

A.7.3 Hypertable

Hypertable 也是一个 Google Bigtable 的克隆，和 HBase 非常相似。项目的发起公司 Zvents 使用着这个系统，每天写入的数据超过 10 亿个单元。Hypertable 的底层分布式文件系统是 HDFS 或 Kosmos（KFS）。Hypertable 使用多版本并发控制（MVCC），允许用户事务在私有的内存空间中执行，直到事务提交之后才对其他客户端可读。

与 Cassandra 和其他 Bigtable 的衍生项目类似，Hypertable 使用 Bloom filter 和 commit log 来最小化磁盘访问、提高性能。

Hypertable 非常适于分析型应用和处理。和其他非关系型解决方案不同，Hypertable 不常用于网站的后端。

- **网站**：http://www.hypertable.org
- **面向**：列
- **创建者**：Hypertable 项目由 Zvents 于 2007 年 2 月启动。
- **实现语言**：C
- **分布化**：是
- **开源**：是

- **schema**：Hypertable 的数据存储在一个多维表中，可以看做是一个统一有序的键-值对列表。数据的键本质上是四维键值（行、列族、列限定符和时间戳）的连接。
- **客户端**：主要通过 C++ Thrift API 交互（Cassandra 也是用 Thrift，并正在转向 Avro），也支持 Java、Python、Ruby、PHP、Perl、Erlang、Haskell、C#、Perl 和 Ocaml 语言。
- **附加特性**：Hypertable 有自己的查询语言，称为 Hypertable 查询语言（HQL）。HQL 基于 SQL 的模型，可以使用你所熟悉的方法来进行查询，比如 `select * from QueryLogByTimestamp WHERE ROW =^'2010-03-27 17:05';`。这个查询看起来和 SQL 非常类似,但格式略有不同。比如“`^=`”操作符的意思是“以……开头”。

与 Voldemort 和 Cassandra（至少是换用 Avro 之前）类似，Hypertable 的客户端序列化也使用 Thrift API。

A.8　多持久化存储系统

本附录里，我们已经接触到了很多风格的持久化存储，有一个突出的情况是：每个都适用于解决一类特定的问题，或特别擅长某一方面，不擅长其他方面。你可能已经听说了“多语言编程”这个概念，这很大程度上归功于 Neal Ford。多语言编程的理念是，不同的编程语言擅长不同的事情，你可以将多个编程语言用在同一个解决方案中，来最大化收益。http://polyglotprogramming.com 的 Dean Wampler 举了 Emacs 的压倒性成功作为例子，说明多语言编程的好处：Emacs 中，使用 C 开发内核，使它更快速，也使用 Lisp 的一个脚本语言变种 Emacs Lisp（ELisp），让它更易于扩展。多语言编程作为一个概念，在过去几年中已经有很多先行者开始实践了。我们经常听说有些大的网络应用的不同部分使用 Scala、Ruby 和 PHP 写成。比如，根据最近的消息，eBay 的架构主要是使用 Java，但搜索引擎是使用 C++ 的。

我认为，在持久化存储方面我们可能面临类似的趋势。NoSQL 的崛起开始对传统发起挑战——曾几何时，我们认为 RDBMS 可以面对所有任务，因为我们只有它。也有些 NoSQL 提倡者建议，用一个或多个 NoSQL 来取代 RDBMS。但我到更乐意见到*多持久化存储系统*的解决方案，或者说，在一个应用里，使用不同的数据存储系统来完成不同的任务。在这个愿景下，关系型数据库将会和非关系型数据库共生，它们会共同出现在模块化、面向服务的应用中，每个都执行它们最适于做的工作。

A.9　小结

在前面的章节中，我们快速浏览了多种非关系型数据库，希望给出恰当的背景来了

解 Cassandra 在 NoSQL 发展中的地位。目的在于了解近年来各种产业是如何考虑数据的，比较这些系统，以了解更广泛的理论基础。

为了了解关系型数据库的替代产品，我们还了解了很多近年来涌现的所谓 NoSQL 系统。这些数据库，具有不同的形式、以不同的方式来应对着正在增长的“互联网规模的”海量数据处理的需求。浏览这些产品的另一个目的是解释 NoSQL 数据库存在的原因，虽然很多反对者和业界的专家都对 NoSQL 颇有微词，但已经有很多大公司在面对强烈的数据需求时采纳了这些 NoSQL 数据库，他们不只是款式新颖的“花瓶”。他们的基础来自于业界最杰出的关于数据可扩展性的理念，有些理念已经存在了数十年了。并且，虽然这些系统有时还有一些（非常公开的）问题，但关系型数据库也是如此。

所以，本书之中，我的目的不是说服你抛弃所有的关系型数据库，并立刻用新方式来取代它们。我只是想帮助你理解这些关系型数据库替代产品的优缺点，并深入了解 Cassandra，这样，当你面临下一个数据问题时，可以直接回来，并选择最合适的工具，而非默认的工具来应对。

词汇表

难以置信，我们真是一大群白痴。我们在字典里找“洗钱”。

——Peter, 电影《上班一条虫》

词汇表里给出了一些使用 Cassandra 时非常有必要了解的概念定义。在 http://wiki.apache.org/cassandra 确实有不少有用的材料，不过第一次看它们也确实很晕，因为每个新名词都需要更多的新名词来解释。很多概念对于初级甚至是中级 Web 开发者和数据库管理员都十分生涩，这里为了便于参考一并给出。词汇表里很多信息重复或扩展了本书中相关章节的内容。

1. 逆熵（Anti-Entropy）[1]

逆熵或称为副本同步，是 Cassandra 为保证不同节点上的数据副本都可以更新到最新版本而采用的机制。

这里解释一下逆熵的工作方式。在主压紧（参见**压紧**）过程中，服务器发起 TreeRequest/TreeResponse 会话，来和相邻节点交换 Merkle 树。Merkle 树是一个列族数据的哈希表示。如果节点间的树不匹配，它们就会重新协商（或修复）数据，以确定它们应该把最终数据确定为什么。树比较验证是 `org.apache.cassandra.service.AntiEntropyService` 类负责的。`AntiEntropyService` 类实现了单例模式，并定义了 `Differencer` 静态类。这个类用于比较两棵树，如果发现任何差异，都会对有差异的区间发起修复过程。

亚马逊的 Dynamo 中使用了逆熵机制，Cassandra 采用了 Dynamo 的模型（参考 Dynamo 论文的 4.7 节）。

译注 1：因为词汇表是字母排序的，为便于查找，这里所有词汇都保留英文原词了。

在 Dynamo 之中，它们使用了 Merkle 树来支持逆熵机制（参见 **Merkle 树**）。Cassandra 也是如此，但两者实现略有不同，在 Cassandra 之中，每个列族有它自己的 Merkle 树，在主压紧过程中，这个树作为一个快照而创建出来，生存期到发送给环上的邻居节点为止。这样的实现可以节省很多磁盘 I/O。

对于如何修复数据，可以参见**读时修复**。

2. 异步写

异步写（asynchronous write）在文档和邮件列表里有时简写为 Async Write。Cassandra 大量使用了 `ExecutorService` 和 `Future<T>` 这类 `java.util.concurrent` 库的组件来把数据写向缓冲区。

3. Avro

Avro（可能）正在取代 Thrift 成为与 Cassandra 进行交互的 RPC 客户端。Avro 是 Apache Hadoop 的一个子项目[2]，由 Hadoop 和 Lucene 的创立者 Doug Cutting 创建。Avro 的功能类似于 Thrift，但它是动态数据序列化库，相对于 Thrift 来说，具有不需要静态生成的优点。另一个迁移向 Avro 的原因是，Thrift 最早是一个 Facebook 创立的项目，之后捐给了 Apache，再之后就没有什么积极的开发活动了。

这个迁移意味着 Cassandra 服务器将从 `org.apache.cassandra.thrift.CassandraServer` 迁移到 `org.apache.cassandra.avro.CassandraServer`。在本书写作的时候，这个迁移尚在进行中，还没有完成。

你可以从 Avro 的官方主页了解更多的信息：http://avro.apache.org。

4. Bigtable

Bigtable 是 Google 于 2006 年开发的面向列的高性能分布式数据库系统，构建在 Google 文件系统（GFS）之上。Cassandra 直接继承自 Bigtable 和亚马逊的 Dynamo。Cassandra 从 Bigtable 继承了很多东西，如稀疏数组和使用 SSTable 存储数据。

Yahoo! 的 HBase 是 Bigtable 的一个克隆。

可以从这里 http://labs.google.com/papers/bigtable.html 读到 Bigtable 的完整论文。

5. Bloom filter

简单地说，Bloom filter 是一个非常快的用于判断一个元素是否属于一个集合的非确定性算法。因为这个算法可能会返回假阳性值，所以是不确定性的，当然，它不会返回假阴性值。Bloom filter 的工作方式是将一个数据集的值映射到一个位数组，或

译注 2：Avro 已经从 Hadoop“毕业”，成为顶级的 Apache 项目了。

是把一个更大的数据集映射到一个摘要字符串中。按照定义，这个摘要会使用一块相对于原始数据量很小的内存来构成。

Cassandra 在键值查询时使用 Bloom filter，以此减少代价高昂的磁盘访问。每个 SSTable 都有一个与之相关联的 Bloom filter，当进行查询时，会在访问硬盘之前查询 Bloom filter。因为 Bloom filter 不会返回假阴性结果，所以，只要过滤器报告说集合里没有某元素存在，那就一定不存在。而如果过滤器说元素存在，那要再进行磁盘访问，确定是否真的存在。

虽然可能出现假阳性结论算是一个缺点，但 Bloom filter 的优点就是它们确实飞快，因为它的空间效率很高，它（不像简单的数组、哈希表或链表一样）无需存储所有的元素。Bloom filter 主要使用内存，这样会减少很多磁盘访问。这样的一个结果就是，假阳性值的数量会随着元素数量增加而增加。

在 Apache Hadoop、Google 的 Bigtable 和 Squid 代理服务器缓存中，都有 Bloom filter 的应用。Bloom filter 是以其发明人 Burton Bloom 命名的。

6. Cassandra

在希腊神话之中，Cassandra（卡珊德拉）是特洛伊国王普利阿摩斯和王后赫卡柏的女儿。因为她非常漂亮，阿波罗神给了她预知未来的能力。但当她拒绝了阿波罗的挑逗之后，他诅咒了她，从此她仍然可以准确预言任何将要发生的事情，但没有人会相信她的话。Cassandra 预见了她的城市特洛伊的覆灭，但却无力阻止。Cassandra 分布式数据库用她命名。

Cassandra 存储系统是一个 Apache 项目，主页位于 http://cassandra.apache.org。该项目 2009 年 1 月成为了一个孵化器项目。它的关键特征包括无中心、弹性、容错、可调一致、高度可用，是为在跨越数据中心的大量普通服务器上存储超大规模数据而设计的。它被用于 Digg、Facebook、Twitter、Cloudkick、Cisco、IBM、Reddit、Rackspace、SimpleGeo、Ooyala、OpenX 等公司中。

Cassandra 最初由 Facebook 开发，用于解决他们的收件箱搜索问题。开发团队由 Jeff Hammerbacher 领导，核心工程师包括 Avinash Lakshman、Karthik Ranganathan 和搜索团队的 Prashant Malik。2008 年 7 月，项目的代码开放到 Google Code 上。2009 年 3 月成为了 Apache 的孵化器项目，次年 2 月 17 日，通过投票成为了 Apache 顶级项目。

Facebook 的 Lakshman 和 Malik 写了一篇名为“一种无中心结构化存储系统”的论文，介绍了 Cassandra 的核心内容。目前这篇论文可以在 http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf 找到。

Avinash Lakshman 还在 2008 年写过一篇博客，描述他们在 Facebook 如何使用 Cassandra：http://www.facebook.com/note.php?note_id=24413138919&id=9445547199&index=9。

很容易明白 Cassandra 数据库为什么如此命名：它的支持者确信，Cassandra 和其他相关的 NoSQL 数据库就是数据库的未来。尽管最终一致性被广泛地使用在很多公司，包括亚马逊、Google、Facebook 以及 Twitter，但仍有很多人对这个模型心存敌意。据推测，将数据库用希腊神话中的先知来命名也是对 Oracle（希腊神话中的“神谕”）数据库开的一个玩笑。

Ran Tavory 开发的 Java 客户端 Hector 是用 Cassandra 的兄弟命名的。

7. Chiton

在古希腊，Chiton 是一种服装，通常是无袖的，男女皆可。Brandon Williams 用 Chiton 命名了他开发的一个开源项目，这个项目是一个基于 Python GTK 的 Apache Cassandra 浏览器。目前托管于 http://github.com/driftx/chiton。

与此相关的一个开源项目，是用 Twisted Python 写的 Cassandra 底层客户端 API 项目 Telephus。这个项目目前托管在 http://github.com/driftx/Telephus。

8. 集群（cluster）

集群就是两个或多个 Cassandra 实例同台献艺。这些实例使用 Gossip 协议相互通信。

新配置一个实例来引入到集群中时，需要先做一些准备工作。首先准备一个**种子节点**，之后指定两个要监听的端口：Gossip 和 Thrift 接口。集群配置好了之后，可以使用 node tool 来验证设置是否正确。

9. 列（column）

列是 Cassandra 数据模型中的最基本的数据表示单位。一列是一个三元组，包括名（有时也称为“键值”）、值和时间戳。一个列的值和时间戳都是由客户端提供的。列名和值的数据类型都是 Java 的字节数组。时间戳的数据类型是基本类型 `long`。为了避免多线程问题，列是不可变的。

列组织在列族之中。

Cassandra 中的列由接口 `org.apache.cassandra.db.IColumn` 定义，其中定义的操作包括读取 `byte[]` 类型的值或读取 `Collection<IColumn>` 类型的子列，还有查询最近修改的时间。

列会按照它们的类型来排序，包括 `AsciiType`、`BytesType`、`LexicalUUIDType`、`LongType`、`TimeUUIDType`、`UTF8Type`。参见**列族**。

10. 列族（column family）

列族的地位大致相当于关系型模型中的表。它是容纳一组有序的列的容器。

因为每个列族都存储在单独的文件中，所以，最好把一次查询需要的数据放在同一个列族里。

你要在 Cassandra 的配置文件中定义列族。还可以提供全局（每个 keyspace 的）值用于行缓存尺寸、键缓存尺寸以及读时修复率。列族可以是下列两种类型之一：`standard` 或 `super`。

参见**列**、Keyspace、**超级列**。

11. 列名（column name）

一行中存储的名 / 值对的“名”那部分。

12. 列值（column value）

一行中存储的名 / 值对的“值”那部分。列值的尺寸会受到主机内存的限制。

13. commit log

commit log 用于所有的 Cassandra 写操作。当进行一个写操作时，首先就会进入到 commit log 之中，这样，即使发生故障也不会丢失数据，之后，值送入 memtable，这样可以通过内存查询，以便获得较高性能。一旦 memtable 满了，数据就会刷写入 SSTable 文件之中。

这项工作由 `org.apache.cassandra.db.commitlog.CommitLog` 类负责，每次写或删除操作，变更会以 `RowMutation` 对象的形式序列化并追加写到 commit log 之中。这些对象组织为 commit log 段。在默认情况下，commit log 的尺寸达到 128 MB 的阈值就会滚动一次，这时会建立一个新的 commit log 文件，接收写操作。这项设置是可调的。

14. 压紧（compaction）

压紧是通过合并大的累积数据文件的方式来释放空间的过程。这个过程大致相当于关系型世界中的表重建。在压紧过程中，合并的数据会被排序，并创建新的索引文件，而且新合并、排序并加索引的数据会写到一个新文件中。

在压紧期间进行的释放空间的操作包括合并键值、和并列、删除墓碑。这个过程由 `org.apache.cassandra.db.CompactionManager` 类负责。`CompactionManager` 实现了一个 MBean 接口，所以它是支持内省的。

Cassandra 中有几种不同类型的压紧操作。

主压紧有两种触发方式：通过节点侦测或是自动进行。节点侦测会向目标节点的相邻节点发送一个 TreeReques 消息。当一个节点收到 TreeRequest 消息之后，会立刻进行一次只读压紧，其目的是验证列族的数据。只读压紧包含如下几步。

(1) 获取列族中的键值分布。
(2) 行数据加入到验证器中之后，如果列族需要验证，就会创建 Merkle 树，并广播到周边节点。
(3) Merkle 树们放在一起，作为一个 `Differencers`（需要验证或比较的树）的列表发送。
(4) 比较过程由 `StageManager` 类进行，这个类负责管理执行任务时的并发问题。在压紧时，`StageManager` 使用一个逆熵阶段。它使用 `org.apache.cassandra.concurrent.JMXEnabledThreadPoolExecutor` 类，在一个单线程内执行压紧程序，并使这个操作可以作为一个 MBean，支持内省机制。

15. 压缩（compression）

返回值压缩是一个未来版中的特性，但 0.6 还不支持。

16. 一致性（consistency）

一致性意味着一个事务不会让数据库进入不合法状态，不会违反完整性约束。一致性是关系型数据库里的事务的关键方面，是 ACID 属性［原子性（atomic）、一致性（consistent）、隔离性（isolated）和持久性（durable）］之一。在 Cassandra 中的一致性程度可以如下衡量：

N = 存放数据副本的节点数
W = 在写操作成功返回之前必须确认写入成功的副本数
R = 在读操作访问数据对象时，需要获得的最少副本数
W + R > N = 强一致性
W + R <= N = 最终一致性

17. 一致性级别（consistency level）

Cassandra 的配置允许你决定在读写操作过程中，集群中多少个副本给出确认或响应可以判定操作是成功的。一致性级别是基于配置文件中指定的**副本因子**的，而不是系统中的节点总数。

有多个可选的一致性级别可以用来进行性能调优。最高性能的级别，一致性级别也最低。对于读写来说，它们的含义有所不同。这在第 7 章中进行了详细介绍。

对于写操作，

- ZERO：写操作将会在一个后台线程中异步完成，无法确保写操作一定成功。这是写数据最快的方式，但对于操作成功的保障也最少。
- ANY：这个级别是在 Cassandra 0.6 中引入的，意味着你可以确信数据至少已经写到一个节点上了，即使是一个提示（参考**提示移交**）也被看做是一个成功的写入。这也是一种相对弱的一致性级别。
- ONE：保证在返回时，数据至少已经写入到一个节点的 commit log 和 memtable 之中了。如果有一个节点响应，这个操作就被认为是成功的。
- QUORUM：一定数量的节点可以就一个操作达成一致。数量定为副本因子 /2+1。所以，如果副本因子是 10，那么有 6 个副本确认操作成功就可以达到法定数量。
- DCQUORUM：选举（quorum）的一个版本，由同一个数据中心里的副本进行投票，在选举的高一致性和在一个数据中心的副本间进行操作的低延时之间进行平衡。
- ALL：保证在返回时副本因子指定数量的节点都接收到数据了。如果某个副本对写操作无响应，则写操作会失败。这个级别有最高的一致性和最差的性能。

对于读操作，

- ONE：当第一个节点响应时，立刻返回该响应的值。在后台进行读时修复。
- QUORUM：查询所有节点。当一部分副本（副本因子 /2+1）返回的时候，把时间戳最新的值返回客户端。
- DCQUORUM：只保证同一个数据中心的节点被查询到。仅当使用机架感知副本放置策略时可用。
- ALL：查询所有节点，并把时间戳最新的记录返回给客户端。这个级别会等待所有节点响应。如果有任何节点没有响应，读操作都会失败。

注意，读操作没有 ZERO 这个一致性级别，因为在读数据时不要求任何节点响应是没有意义的。

18. 数据中心分片策略（data center shard strategy）

参考**副本策略**。

19. 无中心（decentralized）

Cassandra 是无中心的，因为它没有定义主服务器，而是使用对等方式，从而避免了瓶颈和单点失效。无中心对于 Cassandra 非常重要，因为这样就可以更容易地提高或降低集群规模，节点可以随时进入退出集群，而不影响集群运行。

20. 反范式化（denormalization）

在关系型数据库中，有时候会采用反范式化或创建冗余数据，以改善读密集型应用的性能，如在线分析处理（OLAP）应用。而在 Cassandra 之中，反范式化数据更是一种典型情况，因为这样可以改善性能，并使数据结构服务于查询的需要，从而和标准关系型数据库划清界限，关系型数据库的数据结构通常是根据独立的对象模型设计的。

21. 持久性（durability）

数据库的持久性意味着写操作会永久生效，即使服务器崩溃或突然断电也是如此。

Cassandra 通过在 commit log 后面进行追加写来达到持久性。这样就允许服务器避免数据文件中定位的次数。只有 commit log 需要同步写入，对于周期性或批处理的 commit log 都是如此。

当使用一个单服务节点时，Cassandra 并不立刻让存储服务的核心状态和文件同步。这意味着如果服务器在写操作之后马上关机，当服务器重启后，写的结果可能就不会出现了。注意，单服务器节点不建议使用在生产环境中。

参考 Commit Log。

22. Dynamo

2006 年由亚马逊开发，Dynamo 和 Google 的 Bigtable 都是 Cassandra 的主要设计基础。Cassandra 从 Dynamo 继承了键–值存储、对称的对等架构、基于 gossip 的节点发现、最终一致性以及每次操作可调的一致性。

你可以在 http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html 阅读这篇完整论文：“Dynamo：亚马逊的高可用键值存储”。

23. 弹性（elastic）

读写吞吐量可以随着集群中机器数量的增加而线性增长。

24. 最终一致性（eventual consistency）

一致性是一种描述数据在一个更新操作之后的内部完整性的属性。在强一致性数据库中，这意味着一旦客户端完成一个写操作，所有读者都可以立刻看到新值。在最终一致性系统中，数据库一般不会立刻达到一致状态，但最终会达到（这里的“最终”实际也就是毫秒级的时间，它会利用这点时间将新值送到所有副本，具体时间与数据量、节点数和节点的地理分布特性有关）。DNS 就是一个流行的最终一致性架构的例子。最终一致性有时也称为“弱一致性”。

最终一致性在过去几年越来越流行，因为它可以支持极高的可扩展性。虽然传统的完整一致性数据库也能够达到很高的扩展性，但管理开销会很难负担。当然，最终一致性也有一些缺点，比如编程模型会变得更复杂。

虽然 Cassandra 的最终一致性设计是基于亚马逊的 Dynamo 的设计的，但 Cassandra 可能更应该称为“可调”一致性，而不是纯粹的最终一致性。这是因为 Cassandra 允许你在一个范围内配置**一致性级别**，甚至可以要求 Cassandra 在所有副本都可读之前阻塞住操作（这也就是完全一致性）。

其他最终一致性数据存储系统还包括 Riak、Voldemort、MongoDB、Yahoo! 的 HBase、CouchDB、微软的 Dynomite 和亚马逊的 SimpleDB 和 Dynamo。

25. 故障检测（failure detection）

故障检测是在分布式容错系统中，判断哪个节点发生了故障的过程。Cassandra 的故障检测基于一个增量故障检测算法。这种故障检测机制是在 2004 年由日本先进科学技术研究所首先提出的。增量故障检测基于两个基本思路：第一个思路是，故障检测应该是灵活的，这通过将算法与进行监测的应用解耦来实现；第二个思路是，根据对节点发生故障的确信程度，输出一个连续变化的“嫌疑”级别，这种方法的优点是，它将网络环境的波动性考虑在内了。嫌疑级别会基于观测（心跳的采样）来得出一个更加连续的、具有前摄性的指示来判断或强或弱的故障的可能性，而不是一个简单的死活断言。

Cassandra 中的故障检测在 `org.apache.cassandra.gms.FailureDetector` 类中实现。

可以在 http://ddg.jaist.ac.jp/pub/HDY+04.pdf 阅读 Naohiro Hayashibara 等人的 Phi 增量故障检测的论文。

26. 容错性（fault tolerance）

容错性是指一个系统在它的一个或多个组件发生故障的情况下，可以继续提供服务的能力。容错性还可以看做是平滑降质过程，也就是说，如果系统服务性能在故障后发生下降，也只会和故障组件相关。

27. gossip

gossiper 负责保障集群中的所有节点都能得到其他节点的重要状态信息。gossiper 每秒运行一次，来保证即使是故障或没有在线的节点也能第一时间收到节点状态信息。它的工作被设计为可预测的，即使负载迅速增加也是如此。gossip 协议用来支持节点间的键值再均衡和**故障检测**机制。而且，gossip 也是逆熵策略的一个重要部分。

gossiper 以键值对的形式传播信息，gossip 协议会持续给其他节点传播状态信息，直到这些信息被新信息替代。

当一个服务节点启动后，它会把自己注册到 gossiper。更多的信息可以查看 `org.apache.cassandra.service.StorageService` 类。

还可以参考这篇亚马逊关于 gossip 的论文：http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf。

28. Hector

由 Outbrain 的 Ran Tavory 创立的一个开源项目，托管在 GitHub。Hector 是一个用 Java 语言写的 Cassandra 客户端。它封装了 Thrift，提供了 JMX、连接池和故障恢复功能。

29. 提示移交（hinted handoff）

这是一个保证可用性、容错性和平滑降质的机制。如果一个写请求到达 Cassandra，但是负责这部分数据的节点却由于网络分裂、硬件故障或其他原因而不可用，就会创建一个消息“提示”，并交给“移交”另一个活着的节点，并请它在不可用节点恢复时重新进行写操作。这有两个好处：减少了一个节点由于下线而恢复错过的信息所用的时间，并在弱一致性级别中提高了写性能。注意，提示移交在 ONE、QUORUM、ALL 这些一致性级别看来并不当做是一个合格的写操作的确认。但在一致性级别 ANY 中，提示确实当做一个成功的写操作。换句话说，提示写对它们自己本身不可读。

接收提示的节点将会在节点恢复在线后，通过 gossip 很快获知。如果由于某种原因，提示移交无法进行，系统还可以进行读时修复。

30. 键值（key）

参考**行键值**。

31. keyspace

keyspace 是列族的容器，大致对应于关系模型中的数据库，在 Cassandra 中用于区分不同的应用。关系数据库中，数据库是表的集合，而 keyspace 则是列族的有序集合。可以在 Cassandra 的配置文件中定义 keyspace，或用 API 的定义方法来定义它们。当定义一个 keyspace 时，同时还要指定副本因子和副本放置策略。在一个 Cassandra 集群中，可以有一个或多个 keyspace，典型情况是每个应用有一个 keyspace。

参见**列族**。

32. 字典序（lexicographic ordering）

字典序是两个有序笛卡儿集乘积的自然（字母）排序方式。

33. memtable

新近写入数据的内存表达形式。一旦 memtable 写满，就会被刷写到硬盘之中，成为一个 SSTable。

34. Merkle 树

Merkle 树又称为“哈希树”。它的数据结构是一个二叉树，对于表示大数据集的简短摘要。在一个哈希树里，叶子节点是要进行摘要的数据块（通常是文件系统中的文件）。树中的每个父节点都是它的直接子节点的哈希，这可以紧密压紧摘要信息。

Cassandra 中，Merkle 树由 `org.apache.cassandra.utilsMerkleTree` 类实现。

在 Cassandra 中，Merkle 树用于保证对等网络中节点收到的数据块是未被修改和破坏的。它们还用于加密和验证文件和数据传输的内容，Merkle 树还用于 Google Wave 产品之中。Merkle 树得名于其发明者 Ralph Merkle。

35. Multiget

通过列名查询一组键值。

36. Multiget Slice

查询一组键值的一个子集的列。

37. 节点（node）

一个 Cassandra 实例。通常一个 Cassandra 集群会有很多节点，有时，它们作为一个整体称为一个节点环，或者直接叫做“环”。节点可以指集群中的任意 Cassandra 服务器，而“副本”（replica）则特指拥有其他节点某些数据的副本的节点。

38. Node tool

这是指可执行文件 bin/nodetool，用于检测集群的配置是否合理，并可以进行其他各种维护操作。`nodetool` 的命令包括 `cleanup`、`clearsnapshot`、`compact`、`cfstats`、`decommission`、`drain`、`flush`、`info`、`loadbalance`、`move`、`repair`、`ring`、`snapshot [snapshotname]`、`removetoken` 和 `tpstats`。

比如，你可以使用 `nodetool drain` 来阻止 commit log 接收新的写操作。

39. NoSQL

“NoSQL”这个名词，用来描述一些不使用 SQL 或关系模型的数据库。有时解释为“Not *Only* SQL”，用来指出非关系型数据库的鼓吹者并不认为关系型数据库是个坏主意，相反，只是它们不是唯一的数据存储的选择而已。这个名词是 Rackspace 的 Cassandra 追随者 Eric Evans 造出来的，不过，他本人更愿意使用另一个名词“Big Data”（海量数据）来彰显一个事实，就是这类非关系型数据库并非是因为不是什么（SQL 的实现）而定义的，而是因为它们能做什么（处理海量数据负载）而定义在一起的。在我看来，这个名词已经差不多到了它命运的终点了，因为它过于容易混淆了。它试图把各种有不尽相同的目标、设计决策和特性的数据库放到一起来讨论。还是让 Cassandra 是 Cassandra、让 CouchDB 是 CouchDB、让 Riak 是 Riak 吧。

40. 有序分区器（Order-Preserving Partitioner）

这是一类按照键值顺序存放行的**分区器**，将数据物理结构按照排序方式对齐。将列族配置为有序分区器允许使用区间切片，这样 Cassandra 就可以知道哪个键值放在哪个节点上。

这种分区器不同于**随机分区器**，它的优点是可以提供更有效率的区间查询，但是缺点是键值分布不均匀。

有序分区器（OPP）由 `org.apache.cassandra.dht.OrderPreservingPartitioner` 类实现。

有一种特殊的 OPP 称为配页有序分区器（COPP）。它与普通 OPP 非常类似，但数据信息是按照 En/US 字典序方式排序的，而不是字节序。因此，它对于区域设置相关的应用可能有用。

COPP 由 `org.apache.cassandra.dht.CollatingOrderPreservingPartitioner` 类实现。

参考**令牌**。

41. 分区（partition）

作为一个通用名词，分区指网络分裂，是将一个网络分断，使得一个机器无法和另一个直接通信。分区可能由交换机、路由器或是网口的故障导致。考虑一个有五台机器的集群 { A, B, C, D, E }，其中 {A, B} 在一个子网上，{C, D, E} 在另一个子网上。如果连接两个子网的交换机发生了故障，那么就发生了一个网络分区，隔离出了两个子集群 {A, B} 和 {C, D, E}。

Cassandra 是一个具有容错性的数据库，网络分区也是一种考虑在内的故障。因此，即使发生了网络故障，它也可以继续服务，并当分区故障回复后合并数据。

42. 分区器（partitioner）

分区器控制数据在节点间如何分布。要找到一组数据，Cassandra 必须知道哪个节点存储着要查找的范围的值。有三种分区器：随机分区器（这是默认分区器），有序分区器和配页有序分区器。你可以在 storage-conf.xml 或 cassandra.yaml（对于 0.7 版本）文件中的 `<Partitioner>` 元素配置分区器：`<Partitioner>org.apache.cassandra.dht.RandomPartitioner</Partitioner>`。注意：分区器的选择不影响列的排序，只影响行键值的排序。

一旦选择了分区器的类型，就无法在不破坏数据的情况下改变分区器（因为 SSTable 是不可修改的）。参考**有序分区器**和**随机分区器**。

43. 选举（quorum）

多数节点响应了一个操作。这是一个可选的一致性级别。在选举读中，代理节点会等待大多数节点给出相同的值。这会让读操作慢一些，但会保证不会得到过期的数据。

44. 机架感知策略（Rack-Aware Strategy）

参见**副本放置策略**。

45. 随机分区器（random partitioner）

随机分区器使用 `BigIntegerToken` 存放 MD5 哈希值，通过哈希值来决定键值放在环上的具体位置。这样做的好处是，可以让键值很均匀地分布到集群中，不足在于区间查询的效率不高。它是默认的分区器。

参见**分区器**和**有序分区器**。

46. 区间切片（range slice）

查询一个键值区间的列的子集。

47. 读时修复（read repair）

这也是一种保障环上数据一致性的机制。在读操作时，如果 Cassandra 发现某些节点响应的数据和其他节点的处于不一致状态，会在旧节点上进行读时修复操作。读时修复意味着 Cassandra 会对那个节点上的旧数据发起一次写请求，使用之前读请求得到的最新数据更新它。这会从节点上取出所有数据进行合并，然后将合并的数据写回到不同步的节点上。数据一致性检查是通过比较时间戳以及校验和进行的。

进行数据修复的方法来自 org.apache.cassandra.streaming 包。

48. 副本机制（replication）

在一般分布式系统中，副本机制指将数据的多个副本保存在多个机器上，这样如果有一台机器出现故障或由于网络**分区**不可用，集群中仍然有可用数据。缓存是一个简单的副本机制的形式。在 Cassandra 中，副本机制是一种提供高性能、可用性与容错性的手段。

49. 副本因子（replication factor）

Cassandra 提供了一个可配置的副本因子，允许你决定希望付出多少性能来获得更高的一致性。也就是说，读写一致性级别都要基于副本因子，它是在集群中拥有的数据副本的数量。副本因子可以通过配置文件和 API 进行设置。

参见**一致性级别**。

50. 副本策略（replication strategy）

副本策略有时候也叫放置策略，决定了副本的分布方式。第一个副本总会放在拥有对应这块键值区间**令牌**的节点。其余的副本依据可配置的副本放置策略在集群中分布。

Cassandra 使用了“四人帮”的策略模式，允许可置换的副本放置策略。Cassandra 提供了三种可用的策略。选择合适的策略很重要，因为确定了哪个节点负责什么键值范围，也就决定了哪个节点需要接收写操作，这会对不同场景下的效率有重大影响。不同的可置换策略提供了很大的灵活性，你可以根据网络拓扑和需求来调整策略。

副本放置策略都扩展自 org.apache.cassandra.locator.AbstractReplicationStrategy 抽象类。如果你愿意扩展这个类，也可以实现自己的副本放置策略。

副本放置策略是 keyspace 的属性，通过 <ReplicaPlacementStrategy> 元素配置。它们在第 6 章进行了深入的讨论。

51. 行（row）

在列族中，一行是一个有序映射，以列名为键值，列值为值。对于超级列族，一行是一个超级列名到值的映射，其中的值又是子列名到子列值的映射。**行键值**是每行唯一的标识，每行包含了列的名 / 值对。单一行的大小不能超过磁盘空间的限制。

行通过分区器进行排序，分区器的可能类型包括随机分区器、有序分区器和配页有序分区器。

行的定义位于 `org.apache.cassandra.db.Row` 类。

参见**行键值**。

52. 行键值（row key）

有时简称为“键值”，行键值类似于关系模型中一个对象的主键。它代表了标识一行的所有列的方法，是一个任意长字符串。

在 Thrift 接口中，Java 客户端总是假设行键值是 UTF-8 编码的，但是对其他语言的客户端并非如此，可能需要手工将 ASCII 字符串编码为 UTF-8。

53. SEDA

Cassandra 采用了 SEDA（分阶段事件驱动架构，Staged Event-Driven Architecture）来在高并发的情况下获得更大的吞吐能力。SEDA 试图消除线程相关的过度开销。这些开销是由于调度、锁竞争和缓存不命中造成的。SEDA 中，一个任务并不会始终在同一个线程中，这可能会让代码更复杂一些，但是会产生更好的性能。因此，Cassandra 中的很多关键工作，比如读、修改、gossip、memtable 刷写和压紧都是作为一个阶段（SEDA 中的 S）来进行的。一个阶段本质上说是一个独立的事件队列。

当事件到达进入队列，应用提供的事件管理器会被调用。控制器能够为每个阶段根据需求动态地调整线程数量。SEDA 的优势在于更高的并发性和更好的 CPU、磁盘和网络资源的管理。

你可以通过 Matt Welsh、David Culler 和 Eric Brewer 的位于 http://www.eecs.harvard.edu/~mdw/proj/seda 的原文来了解更多关于 SEDA 的信息。

参见**阶段**。

54. 种子节点（seed node）

种子节点是一个已经在 Cassandra 集群中的节点，用于帮助新加入的节点获取数据并开始运行。新加入的节点在开始会和种子节点进行 gossip 通信，获取状态信息，了解节点环的拓扑。集群中可以有多个种子。

55. 切片（slice）

这是一种读查询。使用 `get_slice()` 通过一组列名或一个列名区间进行查询。使用 `get_range_slice()` 返回一个区间的键值的一个子集列。

56. Snitch

Snitch 是 Cassandra 将节点映射到网络中的物理位置的方法。它帮助判断节点相对于其他节点的位置，以便发现和保证有效地路由请求。有几种不同类型的 Snitch。比如，`EndpointSnitch`（或 `RackInferringSnitch`）可以判断两个节点是否在同一个数据中心或是否在同一个机架。这个策略是通过它们的 IP 地址的第二和第三个字段来判断两个节点的数据中心或机架的相对位置的。

而 `DataCenterEndpointSnitch` 允许为机架指定 IP 子网，按照机架所属的数据中心进行分组。

`PropertyFileSnitch` 允许在一个称为 cassandra-rack.properties 的属性文件中指定 IP 地址到机架和数据中心的映射。

Snitch 策略类位于 `org.apache.cassandra.locator` 包。

57. 稀疏（sparse）

在关系型模型中，每个数据类型（表）的每列都必须有值，即使有些时候这个值是空的。与此不同，Cassandra 代表着一种稀疏的或"无 schema"的数据模型，这意味着一个行可以按照需求有更多或更少的列。这样会更有效率。比如一个 1000 × 1000 的表格，类似于一个关系型的表。如果很多单元格是空值，那么这种存储方式是低效率的。

58. SSTable

SSTable 是 Sorted String Table（有序字符串表）的缩写。这个概念是从 Google 的 Bigtable 里借鉴过来的，SSTable 是 Cassandra 中数据在硬盘上的存储方式。这是一个只允许追加写的日志格式。内存中的数据表（memtable）是写入 SSTable 之前，用于数据缓冲和排序的。SSTable 允许高性能地写并可以被压紧。

SSTable 是不可更改的。一旦 memtable 刷写到磁盘上称为一个 SSTable，就无法应用更改了，**压紧**操作仅仅改变它们在磁盘上的表现形式。

要将数据导入或导出 JavaScript 对象标记（JSOM），可以使用 `org.apache.cassandra.tools.SSTableImporter` 类和 `SSTableExporter` 类。

59. 阶段（stage）

作为 Cassandra 的分阶段事件驱动架构（SEDA）的一部分，阶段是一个工作单位的一个包装。一个单独的操作可以流经多个阶段，直至结束，而非在同一个线程中自始至终。

一个阶段包含到达事件队列、事件处理器和相关联的线程池。阶段由一个控制

器所管理，它决定了调度和线程的分配，Cassandra 使用线程池 `java.util.concurrent.ExecutorService` 实现了这种并发模型。要查看阶段是如何工作的，可以看 `org.apache.cassandra.concurrent.StageManager` 类。

还有一些其他操作也作为阶段实现了，包括处理 memtable 的 `ColumnFamilyStore` 类，和 `StorageService` 的一致性管理器。

一个操作可以在一个线程中开始，之后将工作移交给另一个线程。这个移交并不是直接在线程之间，而是在阶段之间进行的。

参见 SEDA。

60. 强一致性（strong consistency）

对于读操作，强一致性意味着如果发现需要进行读时修复，首先进行读时修复，然后返回结果。

61. 超级列（super column）

超级列的值不是字符串，而是一组有名字的其他列的列表，这里称作子列。子列也是有序的，列的数量没有限制。超级列和一般列的不同还在于它们没有关联的时间戳。

超级列是非递归的，也就是说，它们只有一级深度。超级列只能存放其他列的映射，而不能存放超级列的映射。

超级列在 SuperColumn.java 中定义，这个类实现了 `IColumn` 和 `IColumnContainer` 两个接口。这些接口允许进行的操作包括：取出超级列中的所有子列，通过名字取出一个子列，增加子列，删除子列，检查超级列中子列的数量，一级检查子列的最新修改时间。

超级列是 Facebook 对 Google 的 Bigtable 的数据模型的一个改进。

参见**列族**。

62. Thrift

Thrift 是用于与 Cassandra 通信的 RPC 客户端的名字。它可以静态生成一个接口，用于不同语言的序列化，包括 C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、Smalltalk 和 OCaml。正是这个机制允许使用任意上述客户端语言来与 Cassandra 交互。

Thrift 由 Facebook 于 2007 年 4 月实现，并在 2008 年 5 月作为一个孵化器项目捐给了 Apache。截止到本书写作时，Thrift 很可能会被更新更活跃的 Apache 项目 Avro 所取代。Avro 的另一个优点是不需要静态代码生成。

你可以在该项目主页了解 Thrift 的更多信息：http://incubator.apache.org/thrift。

63. 时间戳（timestamp）

Cassandra 中，列值的时间戳是由客户端提供的，所以客户端的时钟同步很重要。时间戳从传统上说是从 Unix 纪元开始时（1970 年 1 月 1 日 0 点）算起的毫秒值。

64. 令牌（token）

环上的每个节点都有一个单独的令牌，用于标明其所拥有的键值范围。根据环上前一个节点的令牌值，你可以指定自己的令牌或让 Cassandra 生成一个。令牌的表示形式由分区器的类型决定。

对于**随机分区器**，令牌是一个在 $0\sim2^{127}$ 的证书，通过对键值进行 MD5 哈希而得。这个令牌位于 `org.apache.cassandra.dht.BigIntegerToken` 类。

对于**有序分区器**，令牌是一个基于键值的 UTF-8 字符串，位于 `org.apache.cassandra.dht.StringToken` 类。

Cassandra 中的令牌都派生自 `org.apache.cassandra.dht.Token` 类。

65. 墓碑（tombstone）

Cassandra 并不会在删除操作之后立刻删除数据。相反，它将数据标记为一个“墓碑”，表示数据已经被删除，但还没被清空。之后，墓碑可以传播到其他副本之上。

墓碑会在主**压紧**时被清理掉。

66. 向量时钟（vector clock）

向量时钟让分布式系统的事件可以成为部分因果有序的。向量时钟保存了一个逻辑时钟的数组，每个对应一个进程，每个进程包含一个时钟的本地副本。为了保存所有进程在一个一致的逻辑状态中，一个进程会把它的时钟发送给其他进程，然后后者会进行更新。为了保证一致性，通常要遵从下列步骤的某些形式。

所有时钟都从 0 开始。每当有一个线程经历了一个事件，它的时钟就会加一。每当一个进程准备发送一条消息，也被看做是一个事件，也会让时钟加一，然后将整个向量与消息一起发送给外部线程。每次一个进程接收一条消息，也会记为一个事件，所以也会为自己的时钟加一。然后比较它的向量和从外部进程进入的消息的向量。它会使用比较出来的最大值更新自己的时钟。

一个向量时钟时间同步策略有可能会在 Cassandra 的未来版本中引入。

67. 弱一致性（weak consistency）

对于读操作，弱一致性通过首先返回结果，然后再进行必要的**读时修复**，从而提升了性能。

关于作者

Eben Hewitt 是一家跨国公司的应用架构总监，负责系统战略和设计工作。他是 Apache Cassandra 项目的一位文档贡献者，同时也是多本技术书籍的作者，其中包括 *Java SOA Cbook*（O'Reilly 出版）。他是 O'Reilly 的 *97 Things Every Software Architect Should Know* 的合著者之一，也是多本软件书籍的技术审阅者。Eben 已经在 IT 业中闯荡了 12 年了，设计并实现过多个领域中的大规模分布式系统，涉及零售业、旅游、政府和互联网接入商。他曾经在亚洲和美国的多个关于 Cassandra、SOA、REST 以及事件驱动架构等方面的业界会议中受邀进行讲演，并就上述话题接受过领先的业内相关网站的多次访谈。你可以在 Twitter 上关注 Eben，他的账号是 @ebenhewitt。

关于封面

《Cassandra 权威指南》封面上是一只绶带鸟。是雀形目、王鹟亚科的一种食虫鸟类。它们是分布最为广泛的一种王鹟亚科的鸟类，从撒哈拉以南的非洲大陆到东南亚，以及很多太平洋岛屿上都有分布。大多数绶带鸟是留鸟，而包括日本绶带鸟和缎鹟在内的其他绶带鸟属于候鸟。

大多数绶带鸟是雌雄二态的，也就是说雌鸟和雄鸟的外观有较大差别。多数雌绶带鸟没有雄鸟那样的靓丽羽毛，雄鸟同时还拥有长长的尾羽，不同种类的绶带鸟的尾羽也有所差别。比如，雄性亚洲绶带鸟的尾羽长度能达到大约 15 英寸。雌性绶带鸟据信会根据尾羽的长度选择配偶。绶带鸟是一夫一妻制的，这让它们的漂亮的颜色和羽毛显得有点不同寻常，因为通常这种强烈的性征展示都发生在那些非一夫一妻制的物种身上。

绶带鸟的分布非常广泛，栖息地包括热带草原、竹林、雨林、落叶林乃至种植园。大多数绶带鸟都依靠它们敏捷的反应和敏锐的视力在飞行中捕食。

封面图片来自 Cassell 的 Natural History 卷四。封面字体是 Adobe ITC Garamond。文本字体是 Linotype Birka，而标题字体是 Adobe Myriad Condensed，代码字体是 LucasFont 的 TheSansMonoCondensed。

版权声明

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。